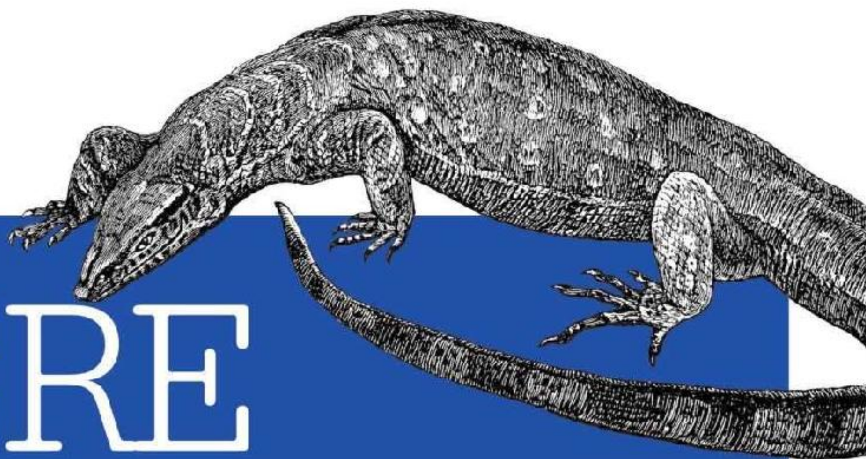
O'REILLY®

Broadview®
www.broadview.com.cn



SRE
Google运维解密

Site Reliability Engineering: How Google Runs Production Systems

[美] Betsy Beyer Chris Jones Jennifer Petoff Niall Richard Murphy 编著

孙宇聪 译

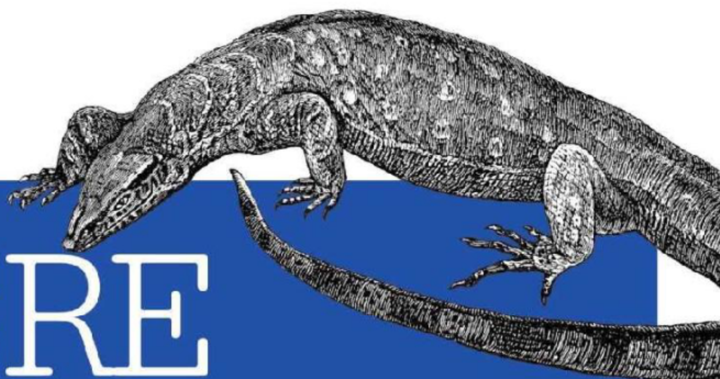
 中国工信出版集团  電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

VISIT...

LANZAROTE Caliente.com

O'REILLY®

Broadview®
www.broadview.com.cn



SRE
Google运维解密

Site Reliability Engineering: How Google Runs Production Systems

[美] Betsy Beyer Chris Jones Jennifer Petoff Niall Richard Murphy 编著

孙宇聪 译

 中国工信出版集团  電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

O·REILIY®

SRE：Google运维解密

Site Reliability Engineering:How Google Runs Production Systems

[美]Betsy Beyer　Chris Jones　Jennifer Petoff　Niall Richard Murphy　编著

孙宇聪　译

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

内容简介

大型软件系统生命周期的绝大部分都处于“使用”阶段，而非“设计”或“实现”阶段。那么为什么我们却总是认为软件工程应该首要关注设计和实现呢？在本书中，Google SRE的关键成员解释了他们是如何对软件进行生命周期的整体性关注的，以及为什么这样做能够帮助Google成功地构建、部署、监控和运维世界上现存最大的软件系统。通过阅读本书，读者可以学习到Google工程师在提高系统部署规模、改进可靠性和资源利用效率方面的指导思想与具体实践--这些都是可以立即直接应用的宝贵经验。

任何一个想要创建、扩展大规模集成系统的人都应该阅读本书。本书针对如何构建一个可长期维护的系统提供了非常宝贵的实践经验。

©2016 by O'Reilly Media,Inc.

Simpli☆ed Chinese Edition,jointly published by O'Reilly Media,Inc.and Publishing House of Electronics Industry,2016.Authorized translation of the English edition,2016 O'Reilly Media,Inc.,the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

本书简体中文版专有出版权由O'Reilly Media,Inc.授予电子工业出版社。未经许可，不得以任何方式复制或抄袭本书的任何部分。专有出版权受法律保护。

版权贸易合同登记号　图字：01-2016-6274

图书在版编目（**CIP**）数据

SRE：Google运维解密/（美）贝特西·拜尔（Betsy　Beyer）等编著；孙宇聪译.—北京：电子工业出版社，2016.10

书名原文：Site Reliability Engineering: How Google Runs Production Systems

ISBN 978-7-121-29726-7

I.①S… II.①贝…②孙… III.①网站—开发 IV.①TP393.092

中国版本图书馆CIP数据核字（2016）第200120号

策划编辑：张春雨

责任编辑：刘舫

封面设计：Karen Montgomery 张健

印刷：北京京科印刷有限公司

装订：北京京科印刷有限公司

出版发行：电子工业出版社 北京市海淀区万寿路173信箱

邮编：100036

开本：787×980 1/16

印张：31

字数：695千字

版次：2016年10月第1版

印次：2016年10月第1次印刷

定价：108.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888,88258888

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至

dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819 faq@phei.com.cn。

O'Reilly Media,Inc.介绍

O'Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项O'Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar博客有口皆碑。”

——Wired

“O'Reilly凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

赞誉

我们都知道Google公司的分布式系统设计和实现在业界遥遥领先，这些分布式系统多年前就已经运行在百万台服务器上，很多公司也都在觊觎这么多服务器是如何运行和管理的。本书揭开了这层神秘的面纱，SRE就是运行和管理这百万台服务器和众多分布式系统的关键。

多年前，Google是通过发布技术论文帮助业界解决分布式难题的，如今各种分布式系统百花齐放，如何管理这些系统对传统的运维技术和理念产生了极大的挑战，现在Google给我们带来了技术指导和最佳实践。该书汇集了Google多年生产环境的管理经验，连编写工作都采用了分布式实现的方法，由各个领域的资深专家联合创作而成。可以把本书看作是一座灯塔，很多公司的集群规模还远达不到Google的规模，但是参照本书中的技术指导和最佳实践，不仅可以加速传统运维向SRE的进化，更重要的是可以帮助公司高效地运维和管理各种复杂的分布式系统。

——吕宏利，Google Ads SRE

信息技术领域是英文缩写词的高产领域，几乎所有的新概念、新技术和新产品的推出甚至一场市场营销的策划都会伴随着新的英文缩写词的出现。SRE这个缩写，在公司内部不仅代表了一个全新的运维理念和其伴随的崭新的工程领域、一套完整的系统运维体系和其对应的最佳实践，而且也是我和我的好朋友——本书的译者孙宇聪一起工作了数年的战斗集体。而本书的作者们也都是这个大集体中的师长和伙伴。

系统运维长久以来都依赖实践积累之上的口口相传，经验通常是领域从业者手里掌握的秘诀。本书从实践出发，汇集了众多业内顶尖的系统运维人员的实战心得，理论基础和实操指导并重，系统化地阐述了在新一代信息系统架构（大规模、分布式、高并发、多业务、多租户）下

系统运维的理念（当前被广泛接受并被大量实践的DevOps就起源于此）、思路、最佳实践以及对应的组织架构和人员管理的方方面面，是系统运维领域从业人员不可多得的参考和学习资料。本书是对新时代系统运维领域实践的总结和理论升华。

本书的译者孙宇聪在生活中是一个略显粗犷的大男人，但对于本书的翻译，他充分发挥了自己在这个领域中多年的从业经验和对系统运维的深刻理解，细致入微地做到内容和语言两个方面的精准和优美，这在翻译的技术图书中是非常难得的。

——张矩，锋瑞资本执行董事，前 Google SRE

很高兴受译者孙宇聪邀请为该书写推荐序，这本书是Google的SRE部门多年实践的总结，孙宇聪本人也在Google SRE部门工作多年。SRE部门在Google真正落实了DevOps。SRE工程师在Google不只是维护各种线上服务的稳定性，还要负责保证各项服务的性能，同时负责管理维护数据中心。美国多家互联网公司都在依照Google的方式来组织和运作SRE部门，可以说SRE被Google发扬光大，Google的SRE实践正在成为DevOps的标准。

SRE和传统的IT运维有很大区别，SRE真正实现了DevOps：首先，SRE深度参与开发阶段的工作，对应用程序的设计实现方式、依赖库、运行时的资源消耗都有严格的规约；其次，SRE工程师本身也要做不少编程工作，来实现各种工具用以自动解决问题和故障，换句话说，SRE强调的是对问题和故障的自动处理，而非人工干预；再者，按照SRE的约定，开发人员自行负责程序上线部署更新，毕竟开发人员对自己开发的程序更熟悉，易于处理程序上线过程中遇到的问题。总之，作为Google的DevOps实践，SRE非常注重开发和运维职能的结合，极大地加快了业务应用迭代周期，提升了IT对业务的支撑能力。

随着DevOps在国内的宣传推广，国内的很多企业客户也逐渐接受了DevOps的理念，但是在具体落地实践DevOps的过程中缺乏实际案例作为参照。本书的推出，方便了国内广大IT人员在落地DevOps过程中参照Google的SRE实践。非常感谢孙宇聪把这么好的一本书翻译成中文。

——王璞，数人云创始人

Google首创了SRE这个职业，并将其SRE思想体系和方法论贡献出来汇集成此书。中文版的及时出版，使得国内广大运维从业者可以更高效地赏阅并实践。很荣幸此书在GOPS全球运维大会首发，高效运维社区将继续作为Google SRE国内第一传播平台，推进其和《互联网应用运维框架及能力模型》（本书译者孙宇聪先生联合撰写）的融合，促进其在中国运维行业的落地生根、蓬勃发展。

——萧田国，高效运维社区发起人，开放运维联盟联合主席

从接触Google SRE的概念开始，就感受到它神秘地存在，直到看到英文版的SRE书籍，才知道它对传统运维的颠覆性。本书的面世，让国内更多的运维人员接触到Google先进的运维理论与实践。个人坚信这种理论和实践的提升与改变，才是运维人的出路，运维的业务价值、行业价值便也随之而来。运维也可以“高大上”地存在！

——王津银，“精益运维”发起人；优维科技创始人；开放运维联盟发起人之一；开放运维联盟应用标准规范组组长、起草人

大型互联网应用的部署规模从几千台到几十万台不一，随着软件系统的复杂度提升也呈现出越来越庞大的趋势，如何通过少数人力管理好庞大复杂的应用环境？如何在环境极度复杂的情况下确保软件的服务质量？如何在确保质量的情况下优化软件迭代速度？很多问题困扰着项目管理者、产品经理、软件工程师、运维人员。本书从Google所面临的问题、价值观、解决方案、体系建设、最佳实践等方面理论结合实际，非常具备指导意义，每一个希望提高工作效率、改进工作成果的技术和管理人员都应该认真阅读理解，结合自身工作环境进行实践，找出一条适合自己的持续发展之路。

——莫显峰，Ucloud联合创始人，CTO

Google丰富的产品与服务已成为全球多数网民每天生活的一部分，而支撑这许多应用的是其背后庞大的基础设施。为了更有效地保证用户体验，Google建立了独树一帜的运维体系并称之为SRE（Site Reliability

Engineering）。绝大部分传统IT公司会雇佣系统管理员（sysadmin）来运维复杂的计算机系统，但由于大部分工作依靠手工操作，所以随着用户增长，Sysadmin的团队也必须相应地增长。Google SRE团队的精华在于研发软件系统，将运维自动化以替代传统模型中的人工操作。这本书详细地描述了Google SRE的原则与理念，并列举了实际案例来说明如何灵活运用这些准则。

孙宇聪在Google任职八年。他不仅精通基础设施的各个方面，还热衷于钻研平台架构。他致力于为中文读者解析Google运维的窍门，于是在繁忙的工作之余，翻译了这本由他的原同事们撰写的书。由于Google的规模很大，许多人可能认为Google的做法无法效仿，但书中描述的原则与道理是可以触类旁通的。书中提及许多实用的道理，比如，100%的可用性是不现实的，需要达到这个目标的成本通常远超于所能获得的价值，所以Google会针对每种产品设定一个错误预算（容错率），既能保证用户体验又不影响创新和部署的速度。

我希望读者像我一样，通过阅读这本书，能学习到如何更有效地运维自己的产品与平台。

——Joe Zhu,Zenlayer创始人

Google SRE 团队通过写作本书为整个运维行业做出了巨大的贡献。通过本书，他们将指导思想、最佳实践和常见的应用架构模式以及团队建设模式共享出来，揭示了 Google如何能够持续不断地建设、部署世界级的工程项目，同时保持世界一流的可靠性标准。每个感兴趣的人都应该通读本书，切身尝试书里提到的一些想法。

Jez Humble,Continuous Delivery和Lean Enterprise书籍的共同作者

我还记得 Google 第一次在运维技术论坛上发表的演讲。感觉就像听了一场野生动物专家针对两栖爬行动物的专题介绍。演讲非常有意思，但是由于演讲的内容和观众的日常工作感觉距离太遥远，因此演讲的效果并不好。

随着IT行业的不断改变，中小型企业的运维实践逐渐和 Google 接

轨。突然之间，Google 多年打磨、积累形成的运维实践变成了最热门的行业焦点。对于一个面临日益严峻的可靠性、可扩展性、可维护性挑战的行业，这本书真是太及时了！

——David N.Blank-Edelman，总监，USENIX董事会成员，以及SREcon 大会的共同创始人

自从我离开 Google 这座充满魔力的城堡，我就一直在等这本书面世，我一直在用书中的思想理念给同事们布道。

——Björn Rabenstein,SoundCloud 生产工程团队负责人，Prometheus（开源项目）开发者，前 Google SRE（2013）

Google 是SRE理念的发明者。本书不光介绍了这个职位的技术细节，还包括了其中的思考过程、团队目标、设计理念以及学到的宝贵课程。如果你想从起源上了解SRE一词的意义，应该从本书开始。

——Russ Allbery,Google SRE，安全工程师

本书的作者们和大家分享了Google SRE团队的成长经历，包括其中走过的弯路。Google 凭借这些实践经验，将Google服务部署到全世界，同时保持世界一流的可靠性。我高度建议任何一个想要创建、扩展大规模集成系统的人阅读本书。这本书针对如何构造一个可长期维护的系统提供了非常宝贵的实践经验。

——Rik Farrow,USENIX成员

开发一个 Gmail 这样的大型分布式系统已经很难了。如何运营维护这样的一套系统，在保障每天不断更新的同时保障一流的可靠性就更难了。这本书就像一套完备的菜谱，收集了Google在实践过程中积累的宝贵经验。希望通过阅读本书，读者能够绕开一些Google曾经走过的弯路。

——Urs Hölzle,Google 基础架构组资深副总裁

译者序

当我在2016年年初听说本书的英文版即将面世时，第一时间就意识到这将是一本不可多得的经典之作。我作为Google SRE曾经的一员，看到本书中提到的那些熟悉的技术和理念时非常兴奋——现在终于有机会用一种体系化、结构化的方式将这些知识和技术与大家分享了！

Google SRE全球共计约1000人，负责运维Google的大部分家喻户晓、不可或缺的商业应用。同时，SRE还负责运维幕后那些全球首屈一指的计算基础设施，不管是全球百万台级别的服务器集群，还是全球一流的网络架构，背后都有SRE的身影。每个小的传统运维问题在这个平台上似乎都被无限放大了。但是与此同时，Google恰恰又是利用最传统、最朴素的软件工程方法将其一一解决的。

SRE是一群天生的怀疑论者，我们怀疑一切宣传起来"高大上"的技术，以及任何"神奇"的产品——我们只想看具体的设计架构、实现细节，以及真实的监控图表。SRE在保障系统可靠性方面并没有什么万能药，有的只是这种极强的务实态度（pragmatic）。这种务实的态度决定了SRE会认真对待运维问题。在设计评审中，他们会认真推演各种灾难场景。在每周例会时，他们又会讨论如何消除和防范事故发生、优化各种警报策略以及增强自动化功能。在平时工作中，他们则会精心维护团队的各种文档和项目源代码，一点一点地提高服务质量。回头看来，SRE其实是一群崇尚工匠主义的人，我们坚信只要不断地解决根源问题，服务质量就一定会得到提升。而SRE正是用这种"日拱一卒"的方法造就了Google这个世界级的奇迹。

本书的风格亦是如此。书中很多章节用务实的语言记录了Google SRE团队在面临各种困难时的思考过程、所采用的解决方案以及事后总结的经验教训。本书中没有介绍任何"魔法系统"，也没有提供任何"奇技淫巧"，有的只是对问题本质发人深省的深入探讨。从这种意义上讲，本

书体系化地覆盖了运维工作的方方面面，是一本运维行业的教科书。我希望通过翻译此书，能将这种体系和理念分享给更多的人。期待与大家更深入地探讨与交流！

回首在Google度过的8年时光，我想感谢我所有的前同事，感谢他们对我的各种帮助，这段职业经历是我终生难忘的。而且，我还要感谢我的家人，是他们的耐心陪伴和帮助才让我踏踏实实地度过了这200多个小时，完成了我人生中最大的一个Project。

孙宇聪

2016年8月3日 傍晚

目录

封面

扉页

版权信息

O'Reilly Media,Inc.介绍

赞誉

译者序

前言

序言

第I部分 概览

- 第1章 介绍
  - 系统管理员模式
  - Google的解决之道：SRE
  - SRE方法论
    - 确保长期关注研发工作
    - 在保障服务SLO的前提下最大化迭代速度
    - 监控系统
    - 应急事件处理
    - 变更管理
    - 需求预测和容量规划

资源部署
效率与性能
小结
第2章 Google 生产环境：SRE视角
硬件
管理物理服务器的系统管理软件
管理物理服务器
存储
网络
其他系统软件
分布式锁服务
监控与警报系统
软件基础设施
研发环境
莎士比亚搜索：一个示范服务
用户请求的处理过程
任务和数据的组织方式
第II部分 指导思想
第3章 拥抱风险
管理风险
度量服务的风险

- 服务的风险容忍度
  - 辨别消费者服务的风险容忍度
  - 基础设施服务的风险容忍度
- 使用错误预算的目的注1
  - 错误预算的构建过程
  - 好处

第4章 服务质量目标

- 服务质量术语
  - 指标
  - 目标
  - 协议
- 指标在实践中的应用
  - 运维人员和最终用户各关心什么
  - 指标的收集
  - 汇总
  - 指标的标准化
- 目标在实践中的应用
  - 目标的定义
  - 目标的选择
  - 控制手段
  - SLO可以建立用户预期

协议在实践中的应用
第5章 减少琐事
琐事的定义
为什么琐事越少越好
什么算作工程工作
琐事繁多是不是一定不好
小结
第6章 分布式系统的监控
术语定义
为什么要监控
对监控系统设置合理预期
现象与原因
黑盒监控与白盒监控
4个黄金指标
关于长尾问题
度量指标时采用合适的精度
简化，直到不能再简化
将上述理念整合起来
监控系统的长期维护
Bigtable SRE：警报过多的案例
Gmail：可预知的、可脚本化的人工干预

长跑
小结
第7章 Google 的自动化系统的演进
自动化的价值
一致性
平台性
修复速度更快
行动速度更快
节省时间
自动化对Google SRE的价值
自动化的应用案例
Google SRE的自动化使用案例
自动化分类的层次结构
让自己脱离工作：自动化所有的东西
舒缓疼痛：将自动化应用到集群上线中
使用Prodtest检测不一致情况
幂等地解决不一致情况
专业化倾向
以服务为导向的集群上线流程
Borg：仓库规模计算机的诞生
可靠性是最基本的功能

建议
第8章 发布工程
发布工程师的角色
发布工程哲学
自服务模型
追求速度
密闭性
强调策略和流程
持续构建与部署
构建
分支
测试
打包
Rapid系统
部署
配置管理
小结
不仅仅只对Google有用
一开始就进行发布工程
第9章 简单化
系统的稳定性与灵活性

乏味是一种美德
我绝对不放弃我的代码
“负代码行”作为一个指标
最小 API
模块化
发布的简单化
小结
第III部分 具体实践
第10章 基于时间序列数据进行有效报警
Borgmon的起源
应用软件的监控埋点
监控指标的收集
时间序列数据的存储
标签与向量
Borg规则计算
报警
监控系统的分片机制
黑盒监控
配置文件的维护
十年之后
第11章 on-call轮值

介绍
on-call工程师的一天
on-call工作平衡
数量上保持平衡
质量上保持平衡
补贴措施
安全感
避免运维压力过大
运维压力过大
奸诈的敌人——运维压力不够
小结
第12章 有效的故障排查手段
理论
实践
故障报告
定位
检查
诊断
测试和修复
神奇的负面结果
治愈

案例分析
使故障排查更简单
小结
第13章 紧急事件响应
当系统出现问题时怎么办
测试导致的紧急事故
细节
响应
事后总结
变更部署带来的紧急事故
细节
事故响应
事后总结
流程导致的严重事故
细节
灾难响应
事后总结
所有的问题都有解决方案
向过去学习，而不是重复它
为事故保留记录
提出那些大的，甚至不可能的问题：假如……

鼓励主动测试

小结

第14章 紧急事故管理

无流程管理的紧急事故

对这次无流程管理的事故的剖析

过于关注技术问题

沟通不畅

不请自来

紧急事故的流程管理要素

嵌套式职责分离

控制中心

实时事故状态文档

明确公开的职责交接

一次流程管理良好的事故

什么时候对外宣布事故

小结

第15章 事后总结：从失败中学习

Google的事后总结哲学

协作和知识共享

建立事后总结文化

小结以及不断优化

第16章 跟踪故障
Escalator
Outalator
聚合
加标签
分析
未预料到的好处
第17章 测试可靠性
软件测试的类型
传统测试
生产测试
创造一个构建和测试环境
大规模测试
测试大规模使用的工具
针对灾难的测试
对速度的渴求
发布到生产环境
允许测试失败
集成
生产环境探针
小结

第18章 SRE部门中的软件工程实践
为什么软件工程项目对SRE很重要
Auxon 案例分析：项目背景和要解决的问题
传统的容量规划方法
解决方案：基于意图的容量规划
基于意图的容量规划
表达产品意图的先导条件
Auxon简介
需求和实现：成功和不足
提升了解程度，推进采用率
团队内部组成
在SRE团队中培养软件工程风气
在SRE团队中建立起软件工程氛围：招聘与开发时间
做到这一点
小结
第19章 前端服务器的负载均衡
有时候硬件并不能解决问题
使用DNS进行负载均衡
负载均衡：虚拟IP
第20章 数据中心内部的负载均衡系统
理想情况

识别异常任务：流速控制和跛脚鸭任务
异常任务的简单应对办法：流速控制
一个可靠的识别异常任务的方法：跛脚鸭状态
利用划分子集限制连接池大小
选择合适的子集
子集选择算法一：随机选择
子集选择算法二：确定性算法
负载均衡策略
简单轮询算法
最闲轮询策略
加权轮询策略
第21章 应对过载
QPS陷阱
给每个用户设置限制
客户端侧的节流机制
重要性
资源利用率信号
处理过载错误
决定何时重试
连接造成的负载
小结

第22章 处理连锁故障
连锁故障产生的原因和如何从设计上避免
服务器过载
资源耗尽
服务不可用
防止软件服务器过载
队列管理
流量抛弃和优雅降级
重试
请求延迟和截止时间
慢启动和冷缓存
保持调用栈永远向下
连锁故障的触发条件
进程崩溃
进程更新
新的发布
自然增长
计划中或计划外的不可用
连锁故障的测试
测试直到出现故障，还要继续测试
测试最常用的客户端

测试非关键性后端
解决连锁故障的立即步骤
增加资源
停止健康检查导致的任务死亡
重启软件服务器
丢弃流量
进入降级模式
消除批处理负载
消除有害的流量
小结
第23章 管理关键状态：利用分布式共识来提高可靠性
使用共识系统的动力：分布式系统协调失败
案例1：脑裂问题
案例2：需要人工干预的灾备切换
案例3：有问题的小组成员算法
分布式共识是如何工作的
Paxos概要：协议示例
分布式共识的系统架构模式
可靠的复制状态机
可靠的复制数据存储和配置存储
使用领头人选举机制实现高可用的处理系统

分布式协调和锁服务
可靠的分布式队列和消息传递
分布式共识系统的性能问题
复合式Paxos：消息流过程详解
应对大量的读操作
法定租约
分布式共识系统的性能与网络延迟
快速Paxos协议：性能优化
稳定的领头人机制
批处理
磁盘访问
分布式共识系统的部署
副本的数量
副本的位置
容量规划和负载均衡
对分布式共识系统的监控
小结
第24章 分布式周期性任务系统
Cron
介绍
可靠性

Cron任务和幂等性

大规模Cron系统

对基础设施的扩展

对需求的扩展

Google Cron系统的构建过程

跟踪Cron任务的状态

Paxos协议的使用

领头人角色和追随者角色

保存状态

运维大型Cron系统

小结

第25章 数据处理流水线

流水线设计模式的起源

简单流水线设计模式与大数据

周期性流水线模式的挑战

工作分发不均造成的问题

分布式环境中周期性数据流水线的缺点

监控周期性流水线的问题

惊群效应

摩尔负载模式

Google Workflow简介

Workflow 是模型—视图—控制器（MVC）模式

Workflow中的执行阶段

Workflow正确性保障

保障业务的持续性

小结

第26章 数据完整性：读写一致

数据完整性的强需求

提供超高的数据完整性的策略

备份与存档

云计算环境下的需求

保障数据完整性和可用性：Google SRE的目标

数据完整性是手段，数据可用性是目标

交付一个恢复系统，而非备份系统

造成数据丢失的事故类型

维护数据完整性的深度和广度的困难之处

Google SRE保障数据完整性的手段

24种数据完整性的事故组合

第一层：软删除

第二层：备份和相关的恢复方法

额外一层：复制机制

1T vs.1E ：存储更多数据没那么简单

第三层：早期预警
确保数据恢复策略可以正常工作
案例分析
Gmail—2011年2月：从GTape上恢复数据（磁带）
Google Music—2012年3月：一次意外删除事故的检测过程
SRE的基本理念在数据完整性上的应用
保持初学者的心态
信任但要验证
不要一厢情愿
纵深防御
小结
第27章 可靠地进行产品的大规模发布
发布协调工程师
发布协调工程师的角色
建立发布流程
发布检查列表
推动融合和简化
发布未知的产品
起草一个发布检查列表
架构与依赖

集成
容量规划
故障模式
客户端行为
流程与自动化
开发流程
外部依赖
发布计划
可靠发布所需要的方法论
灰度和阶段性发布
功能开关框架
应对客户端滥用行为
过载行为和压力测试
LCE的发展
LCE检查列表的变迁
LCE没有解决的问题
小结
第IV部分 管理
第28章 迅速培养SRE加入on-call
新的SRE已经招聘到了，接下来怎么办
培训初期：重体系，而非混乱

系统性、累积型的学习方式

目标性强的项目工作，而非琐事

培养反向工程能力和随机应变能力

反向工程：弄明白系统如何工作

统计学和比较性思维：在压力下坚持科学方法论

随机应变的能力：当意料之外的事情发生时怎么办

将知识串联起来：反向工程某个生产环境服务

有抱负的on-call工程师的5个特点

对事故的渴望：事后总结的阅读和书写

故障处理分角色演习

破坏真的东西，并且修复它们

维护文档是学徒任务的一部分

尽早、尽快见习on-call

on-call之后：通过培训的仪式感，以及日后的持续教育

小结

第29章 处理中断性任务

管理运维负载

如何决策对中断性任务的处理策略

不完美的机器

流状态

将一件事情做好

实际一点的建议

减少中断

第30章 通过嵌入SRE的方式帮助团队从运维过载中恢复

第一阶段：了解服务，了解上下文

确定最大的压力来源

找到导火索

第二阶段：分享背景知识

书写一个好的事后总结作为示范

将紧急事件按类型排序

第三阶段：主导改变

从基础开始

获取团队成员的帮助

解释你的逻辑推理过程

提出引导性问题

小结

第31章 SRE与其他团队的沟通与协作

沟通：生产会议

议程

出席人员

SRE的内部协作

团队构成
高效工作的技术
SRE内部的协作案例分析：Viceroy
Viceroy的诞生
所面临的挑战
建议
SRE与其他部门之间的协作
案例分析：将DFP迁移到F1
小结
第32章 SRE参与模式的演进历程
SRE 参与模式：是什么、怎么样以及为什么
PRR 模型
SRE参与模型
替代性支持
PRR：简单PRR模型
参与
分析
改进和重构
培训
“接手”服务
持续改进

简单PRR模型的演进：早期参与模型
早期参与模型的适用对象
早期参与模型的优势
不断发展的服务：框架和SRE平台
经验教训
影响SRE的外部因素
结构化的解决方案：框架
新服务和管理优势
小结
第V部分 结束语
第33章 其他行业的实践经验
有其他行业背景的资深SRE
灾难预案与演习
从组织架构层面坚持不懈地对安全进行关注
关注任何细节
冗余容量
模拟以及进行线上灾难演习
培训与考核
对详细的需求收集和系统设计的关注
纵深防御
事后总结的文化

将重复性工作自动化，消除运维负载
结构化和理性的决策
小结
第34章 结语
附录A 系统可用性
附录B 生产环境运维过程中的最佳实践
附录C 事故状态文档示范
附录D 事后总结示范
附录E 发布协调检查列表
附录F 生产环境会议记录示范
参考文献
索引
关于编著者
封面介绍

前言

如果用一个词语来描述 Google 的历史，那就是不断地“扩大规模”（scaling up）。Google 的成长经历，是计算机行业中数一数二的成功故事，标志着整个社会向 IT 为中心的商业模式的转变。Google 很早就开始实践 IT 与商业模式的结合，也是向社区推广DevOps 理念的先行者。本书就是由来自公司各个部门，切身参与甚至主导了整个行业转型实践的人写成的。

Google 是在一个系统运维工程师行业转型的阶段发展壮大的。Google 的发展史就像是对传统的系统运维理念发出的革命宣言：我们无法按照传统方式运维Google系统，必须要思考一种新的模式，但是同时我们也没有时间等待其他人验证和支持我们的理论。在Principles of Network and System Administration（参见文献[bur99]）一书的介绍中，我提出了一种观点：系统运维本质上是人与计算机共同参与的一项系统性工程。当时的一些评论者对这种观点表示了强烈的反对：“这个行业还远远没有成熟到可以称为一项工程”。在那时，我甚至对整个运维行业产生了怀疑，认为这个行业在个人英雄主义与神秘色彩中已经永久地迷失了自己，无法前进。但是，这时Google 诞生了，Google 的高速发展将我的预言变成了现实。我之前的定义变成了一个具体的词语：SRE，站点可靠性工程师。我的几个朋友切身参与了这个新职业的创立，用软件工程理念和自动化工具定义了这个行业。一开始，他们显得很神秘，Google 公司内的体验和整个行业也格格不入，Google 太特殊了！ 随着时间的推移，公司内外交流逐渐增多。这本书的目标就是将SRE的一些思考和实践带给整个行业，以促进交流。

在本书中，我们不仅仅展示了 Google 是如何建设维护其富有传奇色彩的大型计算集群的。更重要的是，我们展示了在建设过程中，Google 工程师团队是如何学习、成长、反复修改，最后定义出一套完整

的工具和科技体系的过程。IT 行业大多自我封闭，交流过少，很多从业人员都或多或少地受教条主义的限制。如果Google 工程师团队能克服这个惯性，保持开放的精神，那么我们也能够一起和他们面对 IT 行业内最尖端的挑战。

这本书由一群有共同目标的 Google 工程师写就的短文组成。本书的作者们聚集在同一个公司旗下，为了同一个目标努力，本身就是一件很特别的事情。在本书的各个章节之间经常能看到软件系统的共同点，以及工作模式上的共通之处。我们经常可以从多个角度分析不同的决策选择，了解他们是如何一起解决公司内部多种利益冲突的。这些文章并不是严谨的学术研究论文，而是这些人的切身经历。虽然本书的作者们有着不同的工作目标、写作风格，以及技术背景，但是他们都尝试着去真诚地描述自己遇到的问题和解决的经历。这和 IT 行业内的普遍文风截然不同，风格迥异。有些作者会宣称："不要做A，只有做B才是正确的。"另一些作者会更谨慎，行文更富有哲学性。这其实恰恰代表了整个 IT 行业内不同个性融合的现状。而我们作为读者，作为观察者，并不了解整个Google 的历史，也没有参与到具体的决策过程中，无法全面了解当事人所面临的纠缠不清的挑战，只能带着谦逊的态度远远旁观。在阅读本书的过程中，相信读者一定会产生出许多疑问："他们当时为什么没有选择X?""如果他们选择了Y，结果会是怎样？""如果多年之后回头再看，这个选择会是正确的吗？"这些问题，恰恰是阅读本书的最大收获，因为我们第一次有机会将自己的经历、选择和本书陈列的决策逻辑相互对应，从中发现不足和缺陷。

这本书的成书过程也堪称奇迹。今天，我们能感受到整个行业都在鼓吹厚颜无耻的 "代码拿来主义"（just show me the code）。开源软件社区内部正在形成一种"不要问我问题"的风气，过于强调平等却忽略领域专家的意见。Google 是行业内为数不多的，愿意投入精英力量钻研本质问题的公司，而且这些公司精英很多都有工学博士学位。工具永远只是解决方案中的一个小小组件，用来链接日益庞杂的软件、人和海量的数据。这本书中没有万能药，没有什么东西能解决一切问题，但是这恰恰是本书的宗旨：相比最后的软件结果、架构设计而言，真实的设计过程、作者本身的思考经历更有价值。实现细节永远只是短暂存在的，但

是文档化的设计过程却是无价之宝。有机会了解到这些设计的内幕真是太难得了！

这本书，归根到底，记录了 Google 这个公司的成长经历。书中的很多故事都是由相互重叠的故事组成的，这恰恰说明了扩展一个计算机系统，要比简单按照书本上的标准架构放大困难得多。一个公司的成长，意味着整个公司商业模式和工作模式的扩展，而不是简单的资源扩张。仅此一点，这本书就物超所值了。

自省，是一个在 IT 行业内部并不流行的词语。我们不断重复发明各种系统。很多年以来，只有 USENIX LISA 大会论坛以及其他几个专注于操作系统的会议会讨论一些 IT 基础设施的设计和实现。很多年后的今天，IT 行业已经天翻地覆，但是本书仍然弥足珍贵：它详细记录了Google迈过分水岭时期的全过程。很显然，这些经历没有办法完全复制，也许只能被模仿，但是却可以启发读者，指引未来。本书作者们表现出了极大的真诚，显示了谦逊的风格，以及极强的凝聚力、领导力。这些文章记录了作者们的希望、担忧、成功与失败的经历。我向这些作者们和编者们的勇气致敬，正是这种坦率，让我们能够作为旁观者和后来人，从前人的经历中学习到最宝贵的知识。

Mark Burgess

In Search of Certainty一书作者

Oslo,2016年3月

序言

软件工程有的时候和养孩子类似：虽然生育的过程是痛苦和困难的，但是养育孩子成人的过程才是真正需要花费绝大部分精力的地方。但是，传统软件工程专业花费了很多精力讨论软件的开发过程，而不是其后的维护过程。有统计显示，一个软件系统的40%~90% 的花销其实是花在开发建设完成之后不断维护过程中的。[1]行业内流行的一个说法是：一个系统如果已经开发完成，部署在生产环境上，那么它就是 “稳定的”，就不需要那么多工程师花费精力去优化、维护。我们认为这个说法是错误的。从这个视角出发，我们认为如果软件工程职业主要专注于设计和构建软件系统，那么应该有另外一种职业专注于整个软件系统的生命周期管理。从其设计一直到部署，历经不断改进，最后顺利退役。这样一种职业必须具备非常广泛的技能，但是和其他职业的专注点都不同。Google 将这个职位称为站点可靠性工程师（SRE,Site Reliability Engineering）。

那么，站点可靠性工程师究竟代表着什么呢？的确，这个词语并不能够特别清晰地描述这个职位的意义。基本上每个 Google SRE都会被经常问到这个职位到底代表什么意思，以及他们的日常工作究竟是什么。

将这个词语展开来说：首先，也是最重要的一点，SRE 是工程师（engineer）。SRE 使用计算机科学和软件工程手段来设计和研发大型、分布式计算机软件系统。有的时候，SRE 和产品研发团队共同工作，其他时候我们需要开发这些系统的额外组件：例如备份系统和负载均衡系统等。理想情况下，同时推进这些组件在多个项目中复用。还有的时候，我们的任务是想出各种各样的办法用现有组件解决新的问题。

其次，SRE 的关注焦点在于可靠性。Ben Treynor Sloss,Google负责7×24运维的副总裁，SRE名称的发明者，宣称可靠性应该是任何产品设计中最基本的概念：任何一个系统如果没有人能够稳定地使用，就没有

存在的意义。因为可靠性[2]是如此重要，因此SRE 专注于对其负责的软件系统架构设计、运维流程的不断优化，让这些大型软件系统运行得更可靠，扩展性更好，能更有效地利用资源。但是，SRE并不是无止境地追求完美：当一个系统已经“足够可靠”的时候，SRE 通常将精力转而投入到研发新的功能和创造新的产品中。[3]

最后，SRE 的主要工作是运维在分布式集群管理系统上运行的具体业务服务（service）。不论是遍布全球的存储服务，还是亿万用户赖以工作的 E-mail 服务，还是 Google 最初的 Web 搜索服务。SRE 中的“S”最开始指代的就是google.com的运维服务，因为SRE的第一个工作就是维持网站的正常运转。随着时间的推移，SRE 逐渐接管了 Google 内部绝大部分产品系统，包括 Google Cloud Platform 这类开发者平台，也包括内部的一些非网站类的基础设施系统，例如 Bigtable。

虽然我们在这里将 SRE 的职位定义得比较宽泛，但是在这样一个互联网业务高速发展的时代，这个职位的出现毫不奇怪。同样，虽然在应用系统运营维护的过程中有数不清的重要环节需要关注，我们最关注的是“可靠性”这一点也不奇怪。[4]在Web服务领域里，对服务器端软件的优化和修改是相对可控的，变更管理与生产安全又结合得非常紧密，一种类似于 SRE 的职业早晚会在这个环境下诞生。

虽然 SRE 这个行业是在 Google 内部，从Web 社区中诞生的，但我们认为这个职业对其他团队和组织也有很多值得借鉴的地方。本书是对阐述SRE 发展过程的一次尝试：我们既希望将这些宝贵经验共享给其他相关行业，也希望能从其他行业中汲取知识，从而更好地定义各种角色和术语。为了这个目的，本书将通用的理论、设计理念和思想，与实际的应用工具介绍等分开。在某些需要结合Google内部信息讨论主题的时候，我们相信读者可以进行类比，将书中的理念与自己的实际环境相结合，以便得出更为有效的结论。

本书中也包含了一些对 Google 内部生产环境的介绍，将 Google 内部环境与外部常见的开源类软件相对应。这样可以让本书的一些设计理念与实践的结合度更强，应用起来更容易。

最后，我们当然希望社区内出现更多、更可靠的软件系统。我们知

道，创业企业甚至中型企业经常对如何应用这些理念和技术感到困惑。可靠性就像安全性，越早关注越好。这就意味着一些小型创业公司，在应付日常面临的种种挑战时，也应该抽出一部分精力来面对可靠性这个话题。这与盖房子有些类似，如果一开始将整个地基打好并保持继续修缮，要比盖好房子之后再重新修改设计要容易得多。本书第4部分着重介绍了 SRE 团队如何进行内部培训、如何加强内部沟通等最佳实践，很多都可以直接拿来应用。

对中型企业来说，企业内部可能已经有这样的一组人在做着与SRE非常类似的工作。这些人可能并不叫 SRE 这个名字，甚至可能没有受到管理层的重视。在这样的企业中，提高可靠性最好的办法往往就是去认可这些人的工作，并配备足够的激励机制。在牛顿被世界正式认可为物理学家之前，他经常被称作是最后的炼金术师。而这些专注于可靠性的工程师们，正如当年的牛顿一样，是一个新时代的开拓者。

如果一定要为SRE寻找一个起源的话，谁才能够被称为世界上第一个SRE呢？

我们选择了 Margaret Hamilton,MIT 教授，参与了阿波罗登月计划的软件开发工作。她的工作具有现代SRE的一切特性。[5]用她自己的话来讲："团队文化就是从一切经历中不断学习，包括来自那些我们最意想不到的地方的经历。"

在Apollo 7飞船研发期间的某一天，Margaret 带着她的小女儿 Lauren 一起来到公司。在 Margaret 忙着和组员们在大型计算机上运行飞行模拟测试的时候，她的小女儿偷偷地按下了控制台上的 DSKY 键。整个模拟程序出乎意料地崩溃了，导致整个火箭发射程序意外终止。Margaret 和组员调试后发现，原来Lauren 意外触发了 P01 这段子程序的执行，导致了整个模拟过程的失败。（该子程序是起飞前调试程序，执行时会删除现存的导航信息，如果在火箭飞行过程中执行这段程序，计算机将无法继续维持火箭航线，后果将是灾难性的。）

凭借着 SRE的直觉，Margaret 为项目组提交了一个软件改动，申请在飞行程序中增加一项特殊状态检查，以避免飞行员在飞行过程中意外触发P01程序的执行。但不幸的是，NASA 管理层认为，这项错误发生

的可能性太小，根本不值得为此添加这项修改。于是Margaret 没有能够成功提交这项软件修改。她只能在火箭飞行手册中添加了一段文字，写道："在飞行过程中，请勿触发P01程序。"（当时增加这段文字时，很多NASA工程师都认为这很好笑，认为Margaret 是小题大做，几乎所有人都认为宇航员经过如此长时间的专业训练，是绝对不会犯如此低级的错误的。）

几天后，在Apollo 8飞船执行下一项飞行任务时。宇航员 Jim Lovell、William Anders和Frank Borman 三人执行一个长达四天的飞行计划途中，Jim Lovell 意外地触发了 P01程序的执行。更巧的是，当天正好是美国圣诞节，大部分工程师都休假去了。可想而知，当时NASA的一片混乱状态。这次不是演习，而是人命关天的危急时刻，如果不能及时解决，三名宇航员将永远无法安全返回了。所幸，当时 Margaret 的飞行手册更新中恰恰提到了这种情形，并且提供了重新上传数据以及恢复执行的有效办法，在有限的时间内解决了问题，使任务可以继续进行。

Margaret 曾经说过："无论对一个软件系统运行原理掌握得多么彻底，也不能阻止人犯意外错误。"在这次危机过后，Margaret 之前提交的修改申请很快就被批准了。

虽然Apollo 8的事故发生在几十年前，但是工程师们一定不会对此感到陌生，类似的场景总是在不断重演。希望读者以史为鉴，只有靠着对细节的不懈关注，做好充足的灾难预案和准备工作，时刻警惕着，不放过一切机会去避免灾难发生。这就是SRE 最重要的理念！欢迎加入SRE的大家庭！

如何阅读本书

这本书是由一系列短文组成的，由Google SRE成员和前成员共同写就。相比之下，这本书更像是一本会议文集。本书的每一章都可以作为一个独立部分进行阅读，但是读者也可以根据自己的兴趣选择某些章节重点阅读。（如果本书中引用了某些额外文章，你可以在参考文献中找到。）

读者可以按照任何顺序阅读本书，但是我们推荐从第2章和第3章开始。这两章描述了Google的生产运行环境，以及SRE 是如何系统化认知与量化“风险”的（毕竟 “风险”是SRE最关注的要点）。读者当然也可以选择逐章阅读，本书逻辑上分为以下几个部分：理念性介绍（第II部分）、最佳实践（第III部分）和管理经验（第IV部分）。每一部分都配有简介，并且配有SRE成员以前发表的文章的引用地址。最后，本书配有网站https://g.co/SREBook，其中包括了一些有益读物，希望读者能从中获得阅读的乐趣。

本书的印刷约定

下面是本书中使用的字体约定：

斜体（Italic）

表示新术语、URL、E-mail 地址、文件名及文件扩展名。

等宽字体（Constant width）

用于嵌入一小段程序代码，也用于行内的一些编程术语、函数名、数据库名称、数据结构类型、环境变量和关键词等。

等宽加粗字体（**Constant width bold**）

表示命令或其他应该由用户输入的文本。

等宽斜体（Constant width italic）

表示应该替换为用户提供的值的文本。



这个图标表示提示或建议。



这个图标表示一般说明。



这个图标表示警告或注意。

中文版书中切口以“”表示原书页码，便于读者与英文原版图书对照阅读，本书的索引中列的页码也为英文原版图书中的页码。

使用示例代码

补充材料可以到http://g.co/SREBook下载。

本书对你的工作有所帮助。一般来说，可以在你的程序或者文档中使用本书提供的示例代码。你不必联系我们获得许可，除非你要大量传播代码。例如，从书中抄几块代码编写程序不需要许可；销售或分销O'Reilly随书附带光盘上的示例代码则需要许可；引用本书中的示例代码回答问题不需要许可；将书中大量的示例代码附加到你的产品文档中则需要许可。

我们感谢但不要求注明出处。出处的格式一般包括标题、作者、出版商和ISBN。例如，“Site Reliability Engineering,edited by Betsy Beyer,Chris Jones,Jennifer Petoff,and Niall Richard Murphy（O'Reilly）.Copyright 2016 Google,Inc,978-1-491-92912-4.”。

如果你觉得示例代码的使用不合理或不符合以上的许可权限，请随

时联系我们：permissions@oreilly.com。

permissions@oreilly.com

Safari Books Online



Safari Books Online（www.safaribooksonline.com）是一个按需出版的数字图书馆，出版各种专业的书籍和视频，它们由世界上技术和商业领域的优秀作者撰写。

技术人员、软件开发者、网页设计者及商业和创意专业人士都把Safari Books Online当作科研、问题解决、学习及认证训练的主要资源

Safari Books Online为组织、政府机构和个人提供了大量的产品组合和定价方案

Safari Books Online为组织、政府机构和个人提供了大量的产品组合和定价方案。订阅者可以从一个完全可搜索的数据库中获取成千上万的书籍、培训视频和尚未出版的手稿，这些出版商包括OReilly Media,Prentice Hall Professional,Addison-WesleyProfessional,Microsoft Press,Sams,Que,Peachpit Press,Focal Press,Cisco Press,JohnWiley & Sons,Syngress,Morgan Kaufmann,IBM Redbooks,Packt,Adobe Press,FTPress,Apress,Manning,New Riders,McGraw-Hill,Jones & Bartlett,Course Technology,以及其他数十家出版社。如想了解更多Safari Books Online的信息，请在线访问我们。

如何联系我们

请将对本书的评价和存在的问题通过如下地址告知出版者：

美国：

O'Reilly Media,Inc.

1005 Gravenstein Highway North

Sebastopol,CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C 座807 室（100035）

奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网站，你可以在那里找到关于本书的相关信息，包括勘误列表、示例代码以及其他信息。本书的网站地址是：

http:/bit.ly/site-reliability-engineering

对于本书的评论和技术性的问题，请发送电子邮件到 ：

bookquestions@oreilly.com

关于我们的书籍、课程、会议和新闻的更多信息，请参阅我们的网站http://www.oreilly.com。

在Facebook上找到我们：http://facebook.com/oreilly

在Twitter上关注我们：http://twitter.com/oreillymedia

在YouTube上观看我们：http://www.youtube.com/oreillymedia

致谢

本书全靠作者们和技术作家们的不懈努力才得以面世。我们要特别感谢以下内部评审者，他们提供了非常有价值的反馈：Alex Matey、Dermot Duffy、JC van Winkel、John T。

Ben Lutch和Ben Treynor Sloss 是本书在Google内部的赞助者；他们对这项工作的认可和对分享我们运维大规模服务的认可是本书成书的

关键条件。

我们特别致谢 Rik Farrow,;login：的编辑，他与我们的一些作者紧密合作，在USENIX上预发布了本书的一部分内容。

每章中都注明了本章的作者，我们在这里也想特别致谢一下为每一章提供了反馈、讨论以及评审的人。

第3章：Abe Rahey、Ben Treynor Sloss、Brian Stoler、Dave O'Connor、David Besbris、Jill Alvidrez、Mike Curtis、Nancy Chang、Tammy Capistrant、Tom Limoncelli

第5章：Cody Smith、George Sadlier、Laurence Berland、Marc Alvidrez、Patrick Stahlberg、Peter Duff、Pim van Pelt、Ryan Anderson、Sabrina Farmer、Seth Hettich

第6章：Mike Curtis、Jamie Wilkinson、Seth Hettich

第8章：David Schnur、JT Goldstone、Marc Alvidrez、Marcus Lara-Reinhold、Noah Maxwell、Peter Dinges、Sumitran Raghunathan、Yutong Cho

第9章：Ryan Anderson

第10章：Jules Anderson、Max Luebbe、Mikel Mcdaniel、Raul Vera、Seth Hettich

第11章：Andrew Stribblehill、Richard Woodbury

第12章：Charles Stephen Gunn、John Hedditch、Peter Nuttall、Rob Ewaschuk、Sam Greenfield

第13章：Jelena Oertel、Kripa Krishnan、Sergio Salvi、Tim Craig

第14章：Amy Zhou、Carla Geisser、Grainne Sheerin、Hildo Biersma、Jelena Oertel、Perry Lorier、Rune Kristian Viken

第15章：Dan Wu、Heather Sherman、Jared Brick、Mike

Louer、Štěpán Davidovič、Tim Craig

第16章：Andrew Stribblehill、Richard Woodbury

第17章：Isaac Clerencia、Marc Alvidrez

第18章：Ulric Longyear

第19章：Debashish Chatterjee、Perry Lorier

第20章和第21章：Adam Fletcher、Christoph Pfisterer、Lukáš Ježek、Manjot Pahwa、Micha Riser、Noah Fiedel、Pavel Herrmann、Paweł Zuzelski、Perry Lorier、Ralf Wildenhues、Tudor-Ioan Salomie、Witold Baryluk

第22章：Mike Curtis、Ryan Anderson

第23章：Ananth Shrinivas、Mike Burrows

第24章：Ben Fried、Derek Jackson、Gabe Krabbe、Laura Nolan、Seth Hettich

第25章：Abdulrahman Salem、Alex Perry、Arnar Mar Hrafnkelsson、Dieter Pearcey、Dylan Curley、Eivind Eklund、Eric Veach、Graham Poulter、Ingvar Mattsson、John Looney、Ken Grant、Michelle Duffy、Mike Hochberg、Will Robinson

第26章：Corey Vickrey、Dan Ardelean、Disney Luangsisongkham、Gordon Prioreschi、Kristina Bennett、Liang Lin、Michael Kelly、Sergey Ivanyuk

第27章：Vivek Rau

第28章：Melissa Binde、Perry Lorier、Preston Yoshioka

第29章：Ben Lutch、Carla Geisser、Dzevad Trumic、John Turek、Matt Brown

第30章：Charles Stephen Gunn、Chris Heiser、Max Luebbe、Sam Greenfield

第31章：Alex Kehlenbeck、Jeromy Carriere、Joel Becker、Sowmya Vijayaraghavan、Trevor Mattson-Hamilton

第32章：Seth Hettich

第33章：Adrian Hilton、Brad Kratochvil、Charles Ballowe、Dan Sheridan、Eddie Kennedy、Erik Gross、Gus Hartmann、Jackson Stone、Jeff Stevenson、John Li、Kevin Greer、Matt Toia、Michael Haynie、Mike Doherty、Peter Dahl、Ron Heiby

我们对下列贡献者也表示诚挚的感谢，他们为本书提供了很多素材，仔细评审了每一章，参与了访谈部分，提供了意义重大的资源与专家意见，或者是在其他方面帮助了本书：

Abe Hassan、Adam Rogoyski、Alex Hidalgo、Amaya Booker、Andrew Fikes、Andrew Hurst、Ariel Goh、Ashleigh Rentz、Ayman Hourieh、Barclay Osborn、Ben Appleton、Ben Love、Ben Winslow、Bernhard Beck、Bill Duane、Bill Patry、Blair Zajac、Bob Gruber、Brian Gustafson、Bruce Murphy、Buck Clay、Cedric Cellier、Chiho Saito、Chris Carlon、Christopher Hahn、Chris Kennelly、Chris Taylor、Ciara KamaheleSanfratello、Colin Phipps、Colm Buckley、Craig Paterson、Daniel Eisenbud、Daniel V.Klein、Daniel Spoonhower、Dan Watson、Dave Phillips、David Hixson、Dina Betser、Doron Meyer、Dmitry Fedoruk、Eric Grosse、Eric Schrock、Filip Zyzniewski、Francis Tang、Gary Arneson、Georgina Wilcox、Gretta Bartels、Gustavo Franco、Harald Wagener、Healfdene Goguen、Hugo Santos、Hyrum Wright、Ian Gulliver、Jakub Turski、James Chivers、James O'Kane、James Youngman、Jan Monsch、Jason ParkerBurlingham、Jason Petsod、Jeffry McNeil、Jeff Dean、Jeff Peck、Jennifer Mace、Jerry Cen、Jess Frame、John Brady、John Gunderman、John Kochmar、John Tobin、Jordyn Buchanan、Joseph Bironas、Julio Merino、Julius Plenz、Kate Ward、Kathy

Polizzi、Katrina Sostek、Kenn Hamm、Kirk Russell、Kripa Krishnan、Larry Greenfield、Lea Oliveira、Luca Cittadini、Lucas Pereira、Magnus Ringman、Mahesh Palekar、Marco Paganini、Mario Bonilla、Mathew Mills、Mathew Monroe、Matt D.Brown、Matt Proud、Max Saltonstall、Michal Jaszczyk、Mihai Bivol、Misha Brukman、Olivier Oansaldi、Patrick Bernier、Pierre Palatin、Rob Shanley、Robert van Gent、Rory Ward、Rui ZhangShen、Salim Virji、Sanjay Ghemawat、Sarah Coty、Sean Dorward、Sean Quinlan、Sean Sechrest、Shari Trumbo-McHenry、Shawn Morrissey、Shun-Tak Leung、Stan Jedrus、Stefano Lattarini、Steven Schirripa、Tanya Reilly、Terry Bolt、Tim Chaplin、Toby Weingartner、Tom Black、Udi Meiri、Victor Terron、Vlad Grama、Wes Hertlein、and Zoltan Egyed.

我们非常感谢外部评审者提供的深度反馈：Andrew Fong、Björn Rabenstein、Charles Border、David Blank-Edelman、Frossie Economou、James Meickle、Josh Ryder、Mark Burgess、Russ Allbery。

我们同时特别感谢Cian Synnott，本书的创始团队成员。他在本书成书前离开了Google，但是对本书成书有深远的影响。以及Margaret Hamilton，感谢她允许我们在序言中引用了她的故事。另外，我们还特别感谢Shylaja Nukala，感谢她和她的科技作者们对本书的大力支持，她们的努力是不可或缺的。

本书的编辑们还希望亲自感谢以下人员。

Betsy Beyer：感谢我的祖母（我的个人英雄），她无数次在电话中鼓励我。以及Riba，她帮助我度过了数个无眠的夜晚。同时，当然还要感谢所有参与的SRE，与他们协作无比愉快。

Chris Jones：感谢Michelle，她阻止了我加入海盗这一职业，同时也感谢她在出乎意料的地方找到苹果的能力（玩笑话），同时感谢这些年来所有教授我软件工程学的人。

Jennifer Petoff：感谢我的丈夫Scott，感谢他在我写作本书的两年时间内的大力支持，以及感谢他保障了本书编辑们在“沙漠绿洲”中的充足的糖分供应。

Niall Murphy：感谢 Léan、Oisín和Fiachra，感谢他们这些年来对我的抱怨给予的极高容忍度。感谢Dermot，感谢他提供的内部调动机会。

[1]在这些预测中数据变动的幅度这么大，本身就说明软件工程不是一个非常注重精确性的职业（具体细节参见文献［Gla02］）。

[2]在我们的讨论中，可靠性（reliability）是指某个系统能够在指定环境下，成功持续执行某个功能指定的时间的概率（参见文献[Oco12]中的定义）。

[3]我们主要关注的软件系统是大型网站和类似的Web服务；这里我们不会讨论大型核电站、民航，以及医疗器械或者其他安全性要求极高的系统。然而，在第33章中，我们将自己的方法与这些行业中采用的方法进行了比较。

[4]在这里，我们故意与行业内流行的词语 DevOps进行区分。虽然我们认同基础设施即代码的理念，但我们主要关注的是可靠性。同时，我们也更倾向于将运维的需要直接消除，具体细节参见第7章。

[5]在这个故事之外，她同时也参与推广了“软件工程”（Software Engineer）这个词语。

第I部分 概览

这一部分为SRE具体的工作提供了一些概括性介绍，以及SRE究竟与传统的运维存在哪些不同。

第1章：Ben Treynor Sloss,Google运维团队的高级副总裁，SRE名称的发明者，在这里提供了他对SRE的定义，描述了SRE 究竟与其他类似职位存在哪些不同。

第2章：对 Google 生产环境进行了介绍。本章通过介绍Google组建和运维生产环境的细节，为本书其他部分提到的各种专业术语和系统名词提供铺垫。

第1章 介绍

作者：**Benjamin Treynor Sloss**[1]

编辑：**Betsy Beyer**

不能将碰运气当成战略。

——SRE俗语

大家都知道，计算机软件系统离开人通常是无法自主运行的。那么，究竟应该如何去运维一个日趋复杂的大型分布式计算系统呢？

系统管理员模式

雇佣系统管理员（sysadmin）运维复杂的计算机系统，是行业内一直以来的普遍做法。

这些系统管理员负责将现成的软件组件部署于生产环境中，对外提供某种业务服务。系统管理员的主要工作在于应对系统中产生的各种需要人工干预的事件，以及来自业务部门的变更需求。随着系统变得越来越复杂，组件越来越多，用户流量不断上升，相关的事件和变更需求也会越来越多。于是公司需要招聘更多的系统管理员，来应对日益增多的事件。系统管理员的日常工作与研发工程师相差甚远，通常分属两个不同的部门：开发部（Dev）和运维部（Ops）。

这种模型具有许多优势。对新公司来说，这种模式在行业内具有广泛的应用案例可供参考。市场上具有相关从业经历的人也很多，招聘相对容易。很多第三方工具厂商及系统集成厂商都有现成的工具和软件解决方案帮助一个相对初级的系统管理员团队应对简单的系统维护操作，避免重新发明轮子。

但是，很少有人提及这样做以及相应造成的Dev/Ops分离的团队模型存在一些无法避免的问题。下面我们从两个大的方面来阐述。

1.直接成本。直接成本相对清晰，因为系统管理员团队大部分依赖人工处理系统维护事件以及变更的实施。随着系统复杂度的增加，部署规模的扩大，团队的大小基本与系统负载成线性相关，共同增长。

2.间接成本。研发团队和系统运维团队分属两个部门所带来的间接成本就没那么容易度量了，但是这些间接成本往往大得多。从本质上来说，由于研发团队和运维团队背景各异，技术能力与工具使用习惯上差距巨大，工作目标也截然不同。两个团队对产品的可靠程度要求理解不同，具体执行中对某项操作的危险程度评估与可能的技术防范措施也有截然不同的理解。这些细节上的分歧累积起来，最后逐渐演变成目标与方向上的分歧及形成内部沟通问题，甚至最后上升到部门之间的信任与尊重层面。这样的情形是谁也不愿意见到的，但却是时时上演的。

传统的研发团队和运维团队分歧的焦点主要在软件新版本、新配置的变更的发布速度上。研发部门最关注的是如何能够更快速地构建和发布新功能。运维部门更关注的是如何能在他们值班期间避免发生故障。由于绝大部分生产故障都是由于部署某项变更导致的—不管是部署新版本，还是修改配置，甚至有时只是因为改变了用户的某些行为造成了负

载流量的配比变化而导致故障。这两个部门的目标从本质上来说是互相矛盾的。

极端来说，研发部门想要："随时随地发布新功能，没有任何阻拦"，而运维部门则想要："一旦一个东西在生产环境中正常工作了，就不要再进行任何改动。"由于两个部门使用的语境不同，对风险的定义也不一致。在现实生活中，公司内部这两股力量只能用最传统的政治斗争方式来保障各自的利益。运维团队常常宣称，任何变更上线前必须经过由运维团队制定的流程，这有助于避免事故的发生。例如：运维团队会列出一个非常长的检查清单，历数所有以前曾经出现过的生产事故，要求研发团队在上线任何功能之前必须将所有这些事故模拟一遍，确保不会重现。这个清单通常没有任何标准，每项事故的可重现程度、问题价值并不一定是一致的。而开发团队吃过苦头之后也很快找到了自己的应对办法：开发团队宣称他们不再进行大规模的程序更新，而是逐渐转为功能开关调整、增量更新，以及补丁化。采用这些名词的唯一目的，就是为了绕过运维部门设立的各种流程，从而能更快地上线新功能。

Google的解决之道：SRE

SRE 这种模型是Google尝试着从根本上避免产生这种矛盾的结果。SRE团队通过雇佣软件工程师，创造软件系统来维护系统运行以替代传统模型中的人工操作。

SRE究竟是如何在Google起源的呢？ 其实我的答案非常简单：SRE就是让软件工程师来设计一个新型运维团队的结果。当我在2003年加入Google的时候，我的任务就是领导一个由7名软件工程师组成的"生产环境维护组"。当时，我的整个职业生涯都专注于软件工程，所以很自然，我按照自己最习惯的工作方式和管理方式来组建了这个团队。时过境迁，当年的7人团队已经成长为公司内部1000余人的SRE团队，但是SRE团队的指导理念和工作方式还是基本保持了我最初的想法。

SRE方法论中的主要模块，就是SRE团队的构成。每个SRE团队里基本上有两类工程师。

第一类，团队中 50%～60% 是标准的软件工程师，具体来讲，就

是那些能够正常通过Google软件工程师招聘流程的人。第二类，其他40%～50% 则是一些基本满足Google软件工程师标准（具备85%～99%所要求的技能），但是同时具有一定程度的其他技术能力的工程师。目前来看，UNIX 系统内部细节和1～3层网络知识是Google最看重的两类额外的技术能力。

除此之外，所有的SRE团队成员都必须非常愿意、也非常相信用软件工程方法可以解决复杂的运维问题。Google一直密切关注这两类候选人在招聘通过之后在SRE团队中的表现，但是到目前为止还没有发现他们在工作上和成绩上的显著差异。事实上，由于两类工程师技术背景互补，SRE 团队经常能够寻找到全新的、高效的解决问题的方法。

按照这个标准来招聘和管理SRE团队，我们很快发现SRE团队成员具有如下特点：

（a）对重复性、手工性的操作有天然的排斥感。

（b）有足够的技术能力快速开发出软件系统以替代手工操作。

同时，SRE团队和产品研发部门在学术和工作背景上非常相似。因此，从本质上来说，SRE 就是在用软件工程的思维和方法论完成以前由系统管理员团队手动完成的任务。这些SRE倾向于通过设计、构建自动化工具来取代人工操作。

SRE模型成功的关键在于对工程的关注。如果没有持续的、工程化的解决方案，运维的压力就会不断增加，团队也就需要更多的人来完成工作。传统的Ops团队的大小基本与所服务的产品负载呈线性同步增长。如果一个产品非常成功，用户流量越来越大，就需要更多的团队成员来重复进行同样的事情。

为了避免这一点，负责运维这个服务的团队必须有足够的时间编程，否则他们就会被运维工作所淹没。因此，Google为整个SRE团队所做的所有传统运维工作设立了一个50%的上限值。传统运维工作包括：工单处理、手工操作等。设立这样一个上限值确保了SRE团队有足够的时间改进所维护的服务，将其变得更稳定和更易于维护。这个上限值并

不是目标值。随着时间推移，SRE团队应该倾向于将基本的运维工作全部消除，全力投入在研发任务上。因为整个系统应该可以自主运行，可以自动修复问题。我们的终极目标是推动整个系统趋向于无人化运行，而不仅仅是自动化某些人工流程。当然，在实际运行中，服务规模的不断扩张和新功能的上线已经让SRE够忙了！

Google的经验法则是，SRE团队必须将50%的精力花在真实的开发工作上。那么我们是如何确保每个团队都是这样做的呢？首先，我们必须不断地度量每个团队的工作时间分配。依靠这个数据，SRE管理层会对在开发工作上投入时间不够的团队进行调整。通常，管理层会要求该团队将一些常见的运维工作交还给产品研发部门操作，或者从产品研发部门抽调人力参与团队轮值值班工作。此外，还可以停止该SRE团队的一切新增运维工作。只有管理层主动维护每个SRE团队的工作平衡，我们才能保障他们有足够的时间和精力去进行真正有创造性的、自主的研发工作，同时，这也保障了SRE团队有足够的运维经验，从而让他们设计出切实解决问题的系统。

我们发现 Google SRE 模型在运维大规模复杂系统时有很多优势。由于SRE在调整Google系统的过程中常常直接参与开发、修改代码，SRE文化在公司内部基本代表了一种快速、创新、拥抱变化的文化。实践证明，SRE团队运行、维护、改进一个复杂系统所需要的成员数量与系统部署规模呈非线性增长。而运维同样的系统，用传统的系统管理员模型维护则需要更多数量的人。最后，SRE模型不仅消除了传统模型中研发团队和运维团队的冲突焦点，反而促进了整个产品部门水平的整体提高。因为SRE团队和研发团队之间的成员可以自由流动，整个产品部门的人员都有机会学习和参与大规模运维部署活动，从中获得平时难以获得的宝贵知识。普通的开发人员有多少机会能将自己的程序同时跑在100万个CPU的分布式系统上呢？

虽然SRE模型带来了一些优势，但也存在一些问题。Google面对的一个持久性的难题就是如何招聘合适的SRE。首先SRE要和产品研发部门招聘传统的软件开发工程师竞争。其次，由于SRE要求同时具备多项技能，市场上具有相关从业背景和经验的人就更少了。由于SRE模型也比较新，行业内关于如何建立和维护SRE团队的相关信息并不多。（本

书希望能为改变这种情况而努力。）最后，SRE团队建立之后，由于SRE模型中为了提高可靠性需要采取一些与常规做法违背的做法，所以需要强有力的管理层支持才能推行下去。例如：由于一个季度内的错误预算耗尽而停止发布新功能的决定，可能需要管理层的支持才能让产品研发部门重视起来。

DevOps 还是 SRE ？

DevOps 这个名词是在2008年年末流行起来的，截止到本书写作时（2016年初），这个单词的具体意义仍在不断改变中。这个名词的核心思想是尽早将IT相关技术与产品设计和开发过程结合起来，着重强调自动化而不是人工操作，以及利用软件工程手段执行运维任务等。这些思想与许多SRE的核心思想和实践经验相符合。我们可以认为DevOps是SRE核心理念的普适版，可以用于更广范围内的组织结构、管理结构和人员安排。同时，SRE是DevOps模型在Google的具体实践，带有一些特别的扩展。

SRE方法论

虽然每个SRE团队都有自己的工作流程、优先级定义以及日常工作规范，但是所有的SRE团队都有一套共同的核心方法论。一般来说，SRE团队要承担以下几类职责：可用性改进，延迟优化，性能优化，效率优化，变更管理，监控，紧急事务处理以及容量规划与管理。SRE管理层针对这些内容制定了一套完整的沟通准则和行事规范，这些规范规定了SRE是如何操作Google生产环境的，也规定了SRE如何和产品研发部门、测试部门、最终用户进行有效沟通。这些准则和规范能够帮助每一个SRE部门保持良好的研发和运维工作的平衡。

下面这几小节具体描述了Google SRE的几个核心方法论。

确保长期关注研发工作

上文已经讨论过，Google 将 SRE团队的运维工作限制在50%以内。SRE团队应该将剩余时间花在研发项目上。在实践中，SRE管理人员应该经常度量团队成员的时间分配，如果有必要的话，采取一些暂时性措

施将过多的运维压力转移回开发团队处理。例如：将生产环境中发现的Bug和产生的工单转给研发管理人员去分配，或者将开发团队成员加入到轮值on-call体系中共同承担轮值压力等。这些暂时性措施应该一直持续到运维团队的运维工作压力降低到50%以下为止。在实践中，这种措施实际形成了一个良性循环，激励研发团队设计、构建出不需要人工干预、可以自主运行的系统。只有整个产品部门都认同这个模式，认同50%的安全线的重要性，才会共同努力避免这种情况的发生。

SRE 处理运维工作的一项准则是：在每8 ~ 12小时的on-call 轮值期间最多只处理两个紧急事件。这个准则保证了on-call工程师有足够的时间跟进紧急事件，这样SRE可以正确地处理故障、恢复服务，并且要撰写一份事后报告。如果一次轮值过程中处理的问题过多，那么每个问题就不可能被详细调查清楚，运维工程师甚至没有时间从中学习。如果小规模部署下还无法做到合理报警，规模扩大之后这个情况就会更严重。相对而言，如果一个项目的紧急警报非常少，能够持续稳定运行，那么保持这么多on-call工程师可能就是浪费时间。

所有的产品事故都应该有对应的事后总结，无论有没有触发警报。这里要注意的是，如果一个产品事故没有触发警报，那么事后总结的意义可能更大：因为它将揭示监控系统中的漏洞。事后总结应该包括以下内容：事故发生、发现、解决的全过程，事故的根本原因，预防或者优化的解决方案。Google 的一项准则是“对事不对人”，事后总结的目标是尽早发现和堵住漏洞，而不是通过流程去绕过和掩盖它们。

在保障服务SLO的前提下最大化迭代速度

产品研发部门和SRE之间可以通过消除组织架构冲突来构建良好的合作关系。在企业中，最主要的矛盾就是迭代创新的速度与产品稳定程度之间的矛盾。正如上文所说，其表现形式可能是间接的。在SRE模型中，我们选择正面面对这种矛盾，使用的工具是错误预算。

“错误预算”起源于这样一个理念：任何产品都不是，也不应该做到100% 可靠（显然这并不适用于心脏起搏器和防抱死刹车系统等）。一般来说，任何软件系统都不应该一味地追求100% 可靠。因为对最终用户来说，99.999% 和 100% 的可用性是没有实质区别的（详见附录

A）。从最终用户到服务器之间有很多中间系统（用户的笔记本电脑、家庭WiFi、网络提供商和输电线路等），这些系统综合起来可靠性要远低于 99.999%。所以，在99.999% 和 100%之间的区别基本上成为其他系统的噪声。就算我们花费巨大精力将系统变为100%　可靠也并不能给用户带来任何实质意义上的好处。

如果100%　不是一个正确的可靠性目标，那么多少才是呢？这其实并不是一个技术问题，而是一个产品问题。要回答这个问题，必须考虑以下几个方面：

- 基于用户的使用习惯，服务可靠性要达到什么程度用户才会满意？

- 如果这项服务的可靠程度不够，用户是否有其他的替代选择？

- 服务的可靠程度是否会影响用户对这项服务的使用模式？

公司的商业部门或者产品部门必须建立起一个合理的可靠性目标。一旦建立，“错误预算”就是“1-可靠性目标”。如果一个服务的可靠性目标是99.99%，那么错误预算就是 0.01%。这意味着产品研发部门和SRE部门可以在这个范围内将这个预算用于新功能上线或者产品的创新等任何事情。

错误预算可以用于什么范畴呢？研发团队需要用这个预算上线新功能，吸引新用户。理想情况下，我们应该使用错误预算来最大化新功能上线的速度，同时保障服务质量。这个基本模型建立起来之后，许多常见的战术策略，例如灰度发布、1%　AB测试等就全说得通了。这些战术性手段都是为了更合理地使用整个服务的错误预算。

通过引进“错误预算”的概念，我们解决了研发团队和SRE团队之间的组织架构冲突。SRE团队的目标不再是 “零事故运行”,SRE团队和产品研发团队目标一致，都是在保障业务服务可靠性需求的同时尽可能地加快功能上线速度。这个改动虽小，意义却很大。一次“生产事故”不再是一件坏事，而仅仅是创新流程中一个不可避免的环节，两个团队通过协作共同管理它。

监控系统

监控系统是SRE团队监控服务质量和可用性的一个主要手段。所以，监控系统的设计策略值得着重讨论。最普遍的和传统的报警策略是针对某个特定的情况或者监控值，一旦出现情况或者监控值超过阈值就触发E-mail警报。但是这样的报警策略并不是非常有效：一个需要人工阅读邮件和分析警报来决定目前是否需要采取某种行动的系统从本质上就是错误的。监控系统不应该依赖人来分析警报信息，而是应该由系统自动分析，仅当需要用户执行某种操作时，才需要通知用户。

一个监控系统应该只有三类输出。

紧急警报（alert）

意味着收到警报的用户需要立即执行某种操作，目标是解决某种已经发生的问题，或者是避免即将发生的问题。

工单（ticket）

意味着接受工单的用户应该执行某种操作，但是并非立即执行。系统并不能自动解决目前的情况，但是如果一个用户在几天内执行这项操作，系统不会受到任何影响。

日志（logging）

平时没有人需要关注日志信息，但是日志信息依然被收集起来以备调试和事后分析时使用。正确的做法是平时没人会去主动阅读日志，除非有特殊需要。

应急事件处理

可靠性是MTTF（平均失败时间）和MTTR（平均恢复时间）的函数（参见文献[Sch15]）。评价一个团队将系统恢复到正常情况的最有效指标，就是MTTR。

任何需要人工操作的事情都只会延长恢复时间。一个可以自动恢复的系统即使有更多的故障发生，也要比事事都需要人工干预的系统可用

性更高。当不可避免地需要人工介入时，我们也发现与“船到桥头自然直”的态度相比，通过事先预案并且将最佳方法记录在“运维手册（playbook）”上通常可以使MTTR 降低3倍以上。初期几个万能的工程师的确可以解决生产问题，但是长久看来一个手持“运维宝典”经过多次演习的on-call工程师才是正确之路。虽然不论多么完备的“运维手册”也无法替代人的创新思维，但是在巨大的时间压力和产品压力下，运维手册中记录的清晰调试步骤和分析方法对处理问题的人是不可或缺的。因此，Google SRE将大部分工作重心放在“运维手册”的维护上，同时通过“Wheel of Misfortune”等项目[2]不断培训团队成员。

变更管理

SRE的经验告诉我们，大概 70% 的生产事故由某种部署的变更而触发。变更管理的最佳实践是使用自动化来完成以下几个项目：

- 采用渐进式发布机制。
- 迅速而准确地检测到问题的发生。
- 当出现问题时，安全迅速地回退改动。

这三点可以有效地降低变更给SRE和最终用户带来的时间成本和服务质量的下降。通过将人工因素排除在流程之外，这些操作将不再受到经常发生在人身上的“狼来了”思想以及对大量重复性劳动的关注疲劳所影响。于是，变更执行的速度和安全性同时得到了提高。

需求预测和容量规划

需求预测和容量规划简单来说就是保障一个业务有足够的容量和冗余度去服务预测中的未来需求。这里并没有任何特别的概念，但是我们发现行业内有许多团队根本没有这个意识和计划去满足这个要求。一个业务的容量规划，不仅仅要包括自然增长（随着用户使用量上升，资源用量也上升），也需要包括一些非自然增长的因素（新功能的发布、商业推广，以及其他商业因素在内）。

容量规划有几个步骤是必需的：

● 必须有一个准确的自然增长需求预测模型，需求预测的时间应该超过资源获取的时间。

● 规划中必须有准确的非自然增长的需求来源的统计。

● 必须有周期性压力测试，以便准确地将系统原始资源信息与业务容量对应起来。

因为服务容量对可用性来说是极为重要的，很自然的，SRE应该主导容量规划的过程。同时，这也意味着SRE需要主导资源部署的过程。

资源部署

资源的部署（provisinging）是变更管理与容量规划的结合物。在我们的经验里，资源的部署和配置必须能够非常迅速地完成，而且仅仅是在必要的时候才执行，因为资源通常是非常昂贵的。而且这个部署和配置的过程必须要确保能够正确地执行完毕，否则资源就仍然处于不可用状态。增加现有容量经常需要启动新的实例甚至是整个集群，这通常需要大幅度修改现有的集群配置（配置文件、负载均衡、网络等），同时还要执行一系列测试，确保新上线的容量可以正确地服务用户。因此，新资源的部署与配置是一个相对比较危险的操作，必须要小心谨慎地执行。

效率与性能

高效地利用各种资源是任何赢利性服务都要关心的。因为SRE最终负责容量的部署和配置，因此SRE也必须承担起任何有关利用率的讨论及改进。因为一个服务的利用率指标通常依赖于这个服务的工作方式以及对容量的配置与部署上。如果能够通过密切关注一个服务的容量配置策略，进而改进其资源利用率，这可以非常有效地降低系统的总成本。

一个业务总体资源的使用情况是由以下几个因素驱动的：用户需求（流量）、可用容量和软件的资源使用效率。SRE 可以通过模型预测用户需求，合理部署和配置可用容量，同时可以改进软件以提升资源使用效率。通过这三个因素能够大幅度推动一个服务的效率提升（但是并非全部）。

软件系统一般来说在负载上升的时候，会导致延迟升高。延迟升高其实和容量损失是一样的。当负载到达临界线的时候，一个逐渐变慢的系统最终会停止一切服务。换句话说，系统此时的延迟已经是无穷大了。SRE的目标是根据一个预设的延迟目标部署和维护足够的容量。SRE和产品研发团队应该共同监控和优化整个系统的性能，这就相当于给服务增加容量和提升效率了。[3]

小结

Google SRE代表了对行业现存管理大型复杂服务的最佳实践的一个重要突破。由一个简单的想法“我是一名软件工程师，这是我如何来应付重复劳动的办法”而生，SRE模型已经发展成一套指导思想、一套方法论、一套激励方法和一个拥有广阔空间的独立职业。想要了解更多关于SRE相关的信息，请阅读本书的其余部分。

第2章 Google 生产环境：SRE视角

作者：**JC van Winkel**

编辑：**Betsy Beyer**

Google数据中心与其他传统数据中心和小型服务器集群相比非常不同。这些差异有好处也有坏处，本章将详细讨论Google数据中心建设中遇到的机遇与挑战。同时，本章还介绍了一些后续章节将会反复提到的名词及含义。

硬件

Google的大部分计算资源都存放在自主设计的数据中心中。这些数据中心拥有自己设计的供电系统、制冷系统、网络系统以及计算机硬件（参见文献[Bar13]）。和一般的服务器托管中心相比，每个数据中心的计算机硬件基本上是一致的。[4]为了更好地区分物理服务器和软件服务器的概念，我们在全书中都采用了以下两种说法。

物理服务器（machine）

代表具体的硬件（有时候也代表一个VM 虚拟机）。

软件服务器（server）

代表一个对外提供服务的软件系统。

物理服务器上可以运行任何类型的软件服务器。Google 并不会使用专门的物理服务器运行专门的软件服务器。例如，对Google来说，并不存在一个具体的物理服务器运行我们的邮件系统，而是采用一套集群管理系统进行资源分配，它的名称为Borg。

我们意识到Google将软件服务器与物理服务器区分得如此明确有些不同寻常。通常对一个软件业务来说，软件服务器和物理服务器是紧密相连，密不可分的。但是对Google来说，并不是这样。具体区分的原因和细节相信读者读过下面的章节之后就会有所了解。

图 2-1 描绘了一个典型的Google数据中心的拓扑结构：

- 约10台物理服务器组成了一个机柜（Rack）
- 数台机柜组成一个机柜排（Row）
- 一排或多排机柜组成了一个集群（Cluster）
- 一般来说，一个数据中心（Datacenter）包含多个集群
- 多个相邻的数据中心组成了一个园区（Campus）

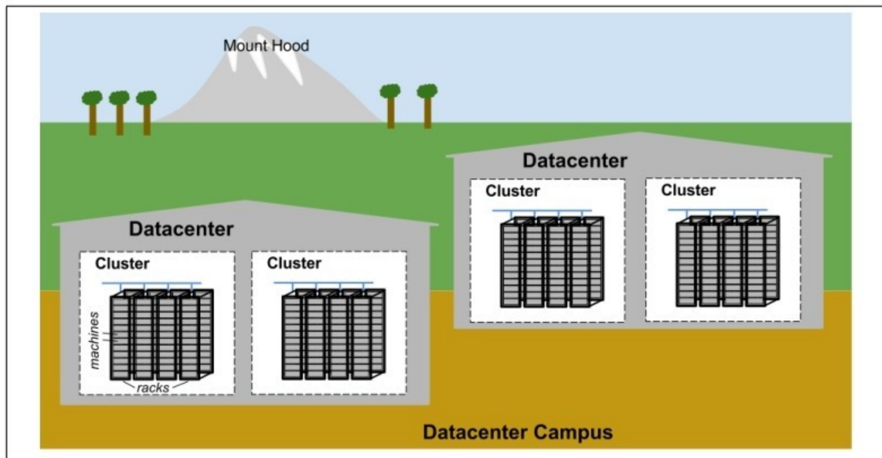


图2-1:Google数据中心园区的拓扑图

每个数据中心内的物理服务器都需要能够互相进行网络通信。为了解决这个问题，我们用几百台Google自己制造的交换机以Clos 连接方式（参见文献[Clos53]）连接起来创建了一个非常快的虚拟网络交换机，这个交换机有几万个虚拟端口。这个交换机的名称叫Jupiter（参见文献[Sin15]）。在Google最大的一个数据中心内，Jupiter可以提供1.3 Pb/s的交叉带宽。

Google的数据中心是由一套全球覆盖的骨干网B4（参见文献[Jai13]）连接起来的。B4 是基于SDN网络技术（使用OpenFlow标准协议）构建的，可以给中型规模的骨干网络提供海量带宽，同时可以利用动态带宽管理优化网络连接，最大化平均带宽（参见文献[Kum15]）。

管理物理服务器的系统管理软件

为了管理和控制硬件设备，我们开发了一套支持大规模部署的系统管理软件。硬件故障是我们用软件系统所解决的一项主要问题。因为一个集群中包括很多硬件设备，每天硬件设备的损坏量很高。在一年内，一个单独集群中平均会发生几千起物理服务器损坏事件，会损失几千块硬盘。当把这些数字乘以Google现有的集群数量时，这些数字就有点令人难以置信了。所以，想将硬件故障与实际业务用户隔离开来。具体业务团队运行软件服务器的时候也并不想每天处理硬件故障。每个数据中心园区都配备专门的团队负责维护硬件设备和数据中心基础设施。

Borg，如图2-2所示，是一个分布式集群操作系统（参见文献[Ver15]）。其与Apache Mesos类似，[5]Borg负责在集群层面管理任务的编排工作。

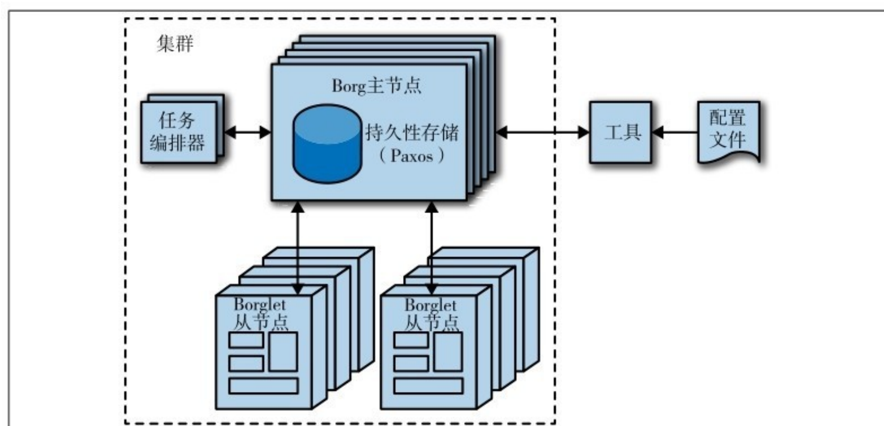


图2-2:Borg 集群管理系统架构抽象图

Borg负责运行用户提交的“任务”（job）。该任务可以是无限运行的软件服务器，或者是批量任务，例如MapReduce（参见文献[Dea04]）。每个任务可以由一个或多个实例（task）组成（有时候甚至由几千个实例组成）。通常这样组织是为了提高冗余度，而且大多数情况下，一个实例并不能满足整个集群的流量需求。当Borg启动一个任务的时候，它会为每一个实例安排一台物理服务器，并且执行具体的程序启动它。Borg 同时会不断监控这些实例，如果发现某个实例出现异常，其会终止该实例，并且重启它，有时候会在另外一台物理服务器上重启。

因为任务实例与机器并没有一对一的固定对应关系，所以我们不能简单地用IP地址和端口来指代某一个具体任务实例。为了解决这个问题，我们增加了一个新的抽象层。每当Borg启动某一个任务的时候，它会给每个具体的任务实例分配一个名字和一个编号，这个系统称之为Borg名称解析系统（BNS）。当其他任务实例连接到某个任务实例时，使用BNS名称建立连接，BNS系统负责将这个名称转换成具体的IP地址和端口进行连接。举例如下，一个BNS地址可能是这样一个字符串：/

bns/＜集群名＞/＜用户名＞/＜任务名＞/＜实例名＞，这个BNS地址最终将会被解析为IP地址：端口。

Borg 还负责将具体资源分配给每个任务。每个任务都需要在配置文件中声明它需要的具体资源（例如：3CPU核心，2GB　内存等）。有了这样的信息，Borg 可以将所有的任务实例合理分配在不同的物理服务器上，以提高每个物理服务器的利用率。同时Borg还关注物理服务器的故障域（failure domain）属性。例如，Borg 不会将某个任务的全部实例都运行在某一个机柜上。因为这样一来，机柜交换机将成为整个任务的单点故障源。

如果一个任务实例资源使用超出了它的分配范围，Borg会杀掉这个实例，并且重启它。我们发现，一个缓慢的不断重启的实例要好过一个永远不重启一直泄露资源的实例。

存储

任务实例可以利用本地硬盘存储一些临时文件，但是我们有几个集群级别的存储系统可供选择作为永久性存储。甚至我们的临时文件存储也在向集群存储模型迁移。这些存储系统和开源的　Lustre，以及Hadoop文件系统（HDFS）类似。

这些存储系统负责向用户提供一套简单易用、可靠的集群存储服务。如图2-3所示，存储系统由多层结构组成：

1.最底层由称为D的服务提供（D代表磁盘　Disk，但是D可以同时使用磁盘和SSD）。D是一个文件服务器，几乎运行在整个集群的所有物理服务器上。然而，用户具体访问某个数据时并不需要记住具体到哪个服务器上去获取，这就是下一层做的事情。

2.D　服务的上一层被称之为Colossus,Colossus建立了一个覆盖了整个集群的文件系统。Colossus文件系统提供传统文件系统的操作接口，同时还支持复制与加密功能。Colossus是GFS的改进版本（参见文献[Ghe03]）。

3.构建于Colossus之上，有几个类似数据库的服务可供选择：

a.Bigtable（参见文献[Cha06]）是一个NoSQL 数据库。它可以处理高达数PB的数据。Bigtable是一个松散存储的、分布式的、有顺序的、持久化的多维映射表。它使用Row　Key、Column　Key以及时间戳做索引。映射表中的值是按原始字节存储的。Bigtable支持“最终一致”的跨数据中心复制模型。

b.Spanner（参见文献[Cor12]）是可以提供 SQL 接口以及满足一致性要求的全球数据库。

c.另外几种数据库系统，例如Blobstore也可用。每一种数据库都有自己的优势与劣势。

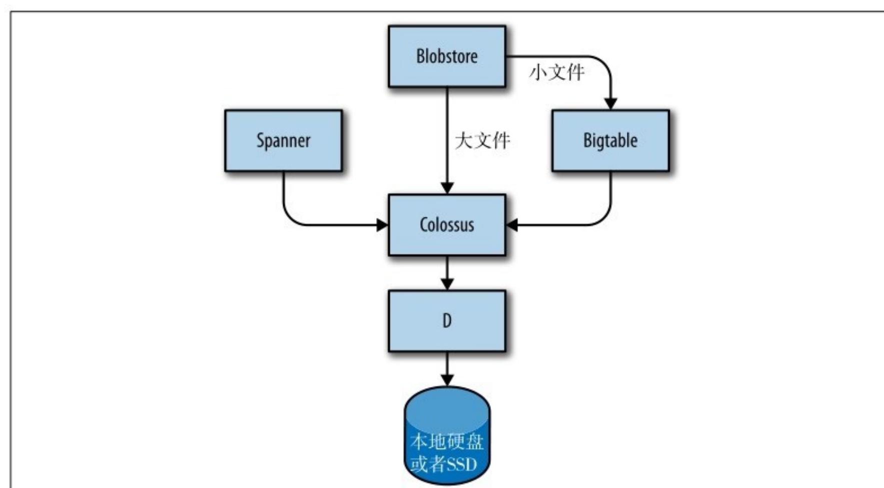


图2-3:Google 存储系统（部分）

网络

Google的网络硬件设备是由以下几种方式控制的。如前文所述，我们使用一个基于OpenFlow协议的软件定义网络（SDN）。我们没有选择使用高级智能路由器，而是采用了普通的非智能交换组件结合集中化（有备份的）的控制器连接方式。该控制器负责计算网络中的最佳路径。我们可以将整个集群的复杂路由计算从具体交换硬件上分离开来，从而降低成本。

网络带宽也需要合理分配。就像Borg给每个任务实例分配计算资源

一样，带宽控制器（Bandwidth Enforcer,BwE）负责管理所有可用带宽。优化带宽的使用的目的不仅仅是降低成本。利用中心化的路由计算，可以解决以前在分布式路由模式下难以解决的流量迁移问题（参见文献[Kum15]）。有些服务包括运行在不同集群上的任务，这些集群通常是遍布全球的。为了降低分布式集群的服务延迟，我们希望能够将用户指派给距离最近的、有空余容量的数据中心处理。我们的全球负载均衡系统（GSLB）在三个层面上负责负载均衡工作：

- 利用地理位置信息进行负载均衡DNS请求（例如www.google.com的解析，具体描述参见第19章）。
- 在用户服务层面进行负载均衡，例如YouTube 和 Google Maps。
- 在远程调用（RPC）层面进行负载均衡（具体描述参见第20章）。

每个服务的管理者在配置文件中给服务起一个名称，同时指定一系列的BNS地址，以及每个BNS地址可用的容量（通常，容量的单位是QPS，每秒请求数量）。GSLB会自动将用户流量导向到合适的位置。

其他系统软件

其他几个运行在数据中心中的组件也很重要。

分布式锁服务

Chubby（参见文献[Bur06]）集群锁服务提供一个与文件系统类似的API用来操作锁。Chubby 可以处理异地、跨机房级别的锁请求。Chubby 使用Paxos协议来提供分布式一致性（具体描述参看第23章）。

Chubby 是实现主实例选举（Master-Election）过程的关键组件。例如，一个服务有5个冗余实例在同时运行，但是只有一个实例可以处理对外请求，一般的做法是通过Chubby进行自动选举，选择一个实例成为主实例。

Chubby适合存放那种对一致性要求非常高的数据。例如BNS服务就

使用Chubby存放BNS路径与IP地址：端口的对应关系。

监控与警报系统

监控系统是服务运维中不可或缺的部分。因此，我们在数据中心中运行了多个Borgmon监控程序实例（具体描述参见10章）。Borgmon定期从监控对象抓取监控指标（Metric）。这些监控指标可以被用来触发警报，也可以存储起来供以后观看。主要有以下几种方式使用监控系统：

- 对真实问题进行报警。
- 对比服务更新前后的状态变化：新的版本是否让软件服务器运行得更快了？
- 检查资源使用量随时间的变化情况，这个信息对合理制定资源计划很有用。

软件基础设施

Google的底层软件基础设施的设计目标是最高效地使用Google的硬件基础设施。我们的代码库大量采用了多线程设计方式，一个任务实例可以同时利用很多个CPU。每个软件服务器都有一个内置的HTTP服务，提供一些调试信息和统计信息，供在线调试、监控等使用。

所有的Google服务之前都使用远程调用（RPC）通信，称为Stubby。我们目前还公布了一个开源实现，gRPC。[6]有时候，一个程序的内部函数调用也会用RPC实现，因为未来有需要时，可以很容易地将其重构为多个组件并行的架构。GSLB 对RPC的负载均衡有良好的支持。

通常来说，一个软件服务器从该服务的前端（frontend）接收RPC请求，同时将另外一些RPC发往该服务器的后端（backend）。一般来说，前端被称为客户端（client），后端被称为服务端（server）。

Protocol Buffer[7]是Google RPC的传输格式，通常简写为

Protobuf，与Apache Thrift类似。Protobuf 相比XML有很多优势，更为简单易用，数据大小比XML格式要小3~10倍，序列化和反序列化速度快100倍，并且协议更加明确。

研发环境

Google非常注重研发效率，我们围绕着自己的基础设施构建了一整套研发环境（参见文献[Mor12b]）。

除了一些开源项目之外（Android和Chrome等），其他Google软件工程师全部使用同一个共享软件仓库开发（参见文献[Pot16]）。这同时也对我们的日常工作流带来一些挑战：

- 如果一个工程师遇到了他工作的项目之外的一个基础组件的问题，他可以直接修改这个问题，向管理者提交一份改动申请（changelist,CL），等待代码评审，最后直接提交。
- 任何对自己项目代码的改动也需要代码评审。

在软件编译的过程中，编译软件会向运行在数据中心的编译服务器发送请求。Google编译软件可以通过并行机制处理超大型编译请求。这套技术架构体系同时也用来进行持续测试。每当一个CL被提交时，所有被这个CL直接或间接影响到的测试都会运行一次。如果测试框架检测到一个CL破坏了其他某个系统的正常工作，测试框架会向提交者发送通知。有些项目组甚至在实践自动部署机制：提交一个新版本，测试通过后，将直接部署于生产环境。

莎士比亚搜索：一个示范服务

为了更好地说明一个服务是怎样利用各种基础设施，以及是如何在Google生产环境中部署的，我们在这里提供一个假想的莎士比亚搜索服务。这个服务的作用是在所有莎士比亚的文献中搜索给定的词语。

整个系统可以分为两大部分：

- 批处理部分（batch）。给全部莎士比亚文献创建索引，同时将索

引写入一个Bigtable中。这项任务只需运行一次（如果发现了新的莎士比亚文献，那就需要再运行一次。）

● 一个应用程序前端服务器（frontend），用以接收处理用户请求。该服务器是一直运行的，因为全球范围的用户都需要使用我们的服务。

批处理部分可以用MapReduce框架完成，三个阶段的实现分别如下所示。

● Mapping 阶段：该程序遍历所有的莎士比亚的文字，将其分成具体的单词。这项任务可以利用多实例并行加速。

● Shuffle 阶段：该程序将上一阶段产生的所有单词和位置等进行排序。

● Reduce 阶段：将上一阶段产生的单词或位置等按单词合并，产生一个新的单词或位置列表。

最后，程序将每一个单词或位置列表写入Bigtable 中，Row Key就是这个单词。

用户请求的处理过程

图2-4 显示了一个用户请求的处理全过程。首先，用户使用浏览器访问https://shakespeare.google.com。为了获得IP地址，用户设备需要请求DNS服务器（1）。该DNS请求最后会到达Google的DNS服务器。Google的DNS 服务器会请求GSLB系统。GSLB 通过全局流量负载信息，决定使用哪个IP地址回复用户。

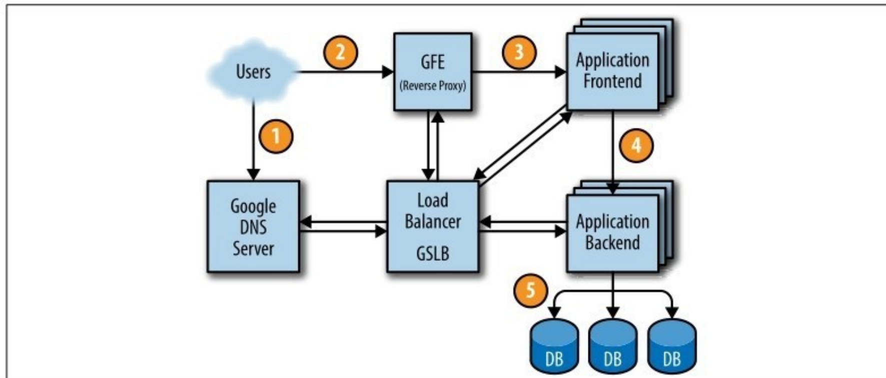


图2-4：用户请求处理过程

用户浏览器利用获得的IP地址连接到HTTP服务器，这个服务器（Google前端服务器GFE）负责终结TCP 连接，并且反向代理请求到真正的服务器上（2）。GFE 从配置文件中找到该请求对应的后端服务（配置文件中包括所有的Google服务，例如Web Search、maps以及本例中的 Shakespeare）。GFE再次咨询GSLB系统，获得一个GSLB分配的、目前可用的Shakespeare服务器地址，向其发送一个RPC请求（3）。

Shakespeare前端服务器分析接收到的请求，构建出一个具体查询的Protobuf 请求。这时Shakespeare前端服务器需要联系后端服务器来做具体查询。前端服务器也需要联系GSLB 服务，获取目前可用的（同时符合负载均衡条件的）后端服务器的BNS地址（4）。Shakespeare 后端服务器随后请求Bigtable 服务器来获得最终查询结果（5）。

最终结果被写入一个Protobuf 结构体中，返回给Shakespeare后端服务器，后端服务器将其回复给Shakespeare前端服务器，前端服务器最终根据这个数据结构构建HTML回复给最终用户。

上述这些连锁事件其实一共耗时几百毫秒。因为一个请求涉及很多组件，这些组件都必须相当可靠才行，GSLB 服务如果出现问题将会造成严重故障。但是Google 依靠严格的测试和灰度发布流程，以及很多主动优雅降级的措施，使得我们可以为用户提供一个非常稳定的服务。由于Google的可靠性举世闻名，人们经常通过访问www.google.com来验证他们的网络服务是否正常。

假设压力测试的结果显示，我们的服务器可以每秒处理大概100个请求（100 QPS）。通过对用户进行的调查显示，我们预计峰值流量会达到 3470 QPS，为了处理这些流量，至少需要 35个任务实例。但是，由于以下几点考量，我们最终决定至少采用37个实例，也就是 N+2模式：

- 在更新过程中，有一个任务实例将会短暂不可用，只有36个实例可提供服务。

- 如果另外一个物理服务器同时也出现问题，那么另外一个任务实例也受到影响，只剩35个实例可以对外服务，刚好可以满足峰值要求。[8]

假设，对用户流量的进一步观察显示我们的峰值流量其实是来自全球的。北美洲1430 QPS，南美洲 290 QPS，欧洲和非洲 1400 QPS，亚洲及澳大利亚共350 QPS。为了更好地服务用户，我们需要将服务分别部署在美国、南美洲、欧洲和亚洲。在南美洲，我们选择使用只部署4个实例（而不是5个），将冗余度降低为N+1。这样做的原因是，我们选择在极端情况下牺牲一些用户体验以降低成本。因为当容量不足时，GSLB会将南美洲的用户流量导向其他可用的数据中心，可以节省大概20%的硬件资源。在有条件的地方，我们还会将任务实例分散在不同的集群中，以便更好地提升可靠性。

因为Shakespeare后端服务器需要连接Bigtable服务读取数据，我们同时也需要合理地安排数据存储。亚洲的后端服务器尝试访问部署在美国的Bigtable会面临延迟问题。所以我们在每个地理区域都存放了Bigtable的副本。

利用Bigtable的复制功能，我们可以同时达到两个目的：

- 当Bigtable服务出现问题时，可以利用副本提供服务。

- 任务实例可以利用本地Bigtable 加速数据访问。

虽然Bigtable仅仅提供最终一致性保障（eventual consistency），但是由于数据更新并不频繁，所以对我们来说这并不是问题。

我们在这一章中介绍了很多术语，在接下来的章节中还会重复引用它们，所以读者并不一定现在就将它们完全记住。

[1]Google VP,Google SRE的创始人。

[2]一个Google内部模拟灾难恢复的演习项目，参见第28章的“故障处理分角色演示”一节。

[3]第31章的“沟通：生产会议”一节会详细讨论这是如何发生的。

[4]大部分情况下，Google数据中心的硬件配置是统一的，但是也有例外情况，例如有些数据中心可能会同时存在几种不同迭代周期产生的硬件产品，我们有时候也会修改某个现有数据中心的硬件配置。

[5]有些读者可能了解Borg的下一代，Kubernetes，开源容器化集群编排系统，Google创立于2014年。请参看 http://kubernetes.io和文献[Bur16]。有关更多Borg与Apache Mesos的共同点的信息，请参看文献[Ver15]。

[6]具体信息参见http://gprpc.io。

[7]Protocol Buffer是编程语言中性的、运行平台中性的一种数据序列化机制。更详细的内容可参见https://developer.google.com/protocol-buffers/。

[8]我们假设同时出现两个任务实例不可用的情况的可能性很低，足以忽略不计。但是单点故障源，例如供电设施问题，或者机柜交换机问题，可能会影响这里的假设。

第II部分 指导思想

本部分将描述SRE日常工作背后的指导思想—工作模式、行为方式，以及平时运维工作中关注的重点等。

本部分的第一章（第3章）是最重要的一章。这一章从最广泛的角度描述了SRE的日常工作，以及背后的指导思想。这一章从“风险”入手，描述了如何评估风险、管理风险，以及利用错误预算的手段来推进中立性的服务运维。

服务质量目标（SLO）是SRE 的另外一个基本概念。运维行业经常会将一系列离散的概念都归结为服务质量协议（SLA），这样使得讨论变得很复杂。第4章试图将SLO与SLA区分开来，详细描述SRE是如何区分这两个术语的，同时针对应用程序性能指标的选择提供了一些建议。

消除琐事（toil）是SRE的一项重要工作，详情请参见第5章。我们将琐事定义为无聊、重复性的运维工作，这些工作通常不具有长期价值，而且会随着服务规模的扩大而增长。

对Google或者其他任何一个公司来说，监控系统都是运维生产环境必不可少的组件。如果没有针对服务的监控，就无从得知目前服务的状态，如果不知道服务的状态，就无从谈起维护服务的可靠性。第6章描述了监控的手段和目标，以及一些与具体实现无关的最佳实践。

第7章描述了Google SRE进行自动化工作的方法论。这一章同时讨论了SRE在自动化过程中的一些成功和失败的案例。

大部分公司不太重视发布工作。然而，在第8章中，我们可以看到，发布工作是整体系统稳定性的一个关键环节，因为大部分故障都是由于新的变更引起的。在这方面的投入也可以保障每次发布的顺利进行。

广义软件工程中（不仅仅是运维部分）的一个关键思想是保持简单。这个特性一旦失去，就很难找回来了。但是不管如何，随着陈旧组件的不断下线，原来复杂的系统一定会逐渐简化。在第9章，我们详细讨论了这个主题。

其他**Google SRE**阅读材料

在保障安全的前提下，提升产品迭代速度是所有组织都想达到的目标。在Make Push On Green a Reality（参见文献[Kle14]，发布于2014年10月）中，我们描述了从发布环节中将人工操作去除，这可以降低SRE需要做的琐事，同时可以增加系统的可靠性。

第3章 拥抱风险

作者：**Marc Alvidrez**

编辑：**Kavita Guliani**

你可能认为Google会试图构建一个百分之百可靠的服务。事实证明，超过一定值后，再提高可靠性对于一项服务（和它的用户）来说，结果可能会更差而不是更好！极端的可靠性会带来成本的大幅提升：过分追求稳定性限制了新功能的开发速度和将产品交付给用户的速度，并且很大程度地增加了成本，这反过来又减少了一个团队可以提供的新功能的数量。此外，用户通常不会注意到一项服务在高可靠性和极端可靠性之间的差异，因为用户体验主要是受较不可靠的组件主导，例如手机移动网络或者他们正在使用的设备。简单地说，用户在一个有着99%可靠性的智能手机上是不能分辨出99.99%和99.999%的服务可靠性的区别的！

基于这一点，SRE旨在寻求快速创新和高效的服务运营业务之间的风险的平衡，而不是简单地将服务在线时间最大化。这样一来，我们可以优化用户的整体幸福感，平衡系统的功能、服务和性能。

管理风险

不可靠的系统会很快侵蚀用户的信心，所以我们想要减少系统出故障的几率。然而，经验表明，在构建系统的过程中，可靠性进一步提升的成本并不是线性增加的—可靠性的下一个改进可能比之前的改进成本增加100倍。高昂的成本主要存在于以下两个维度。

冗余物理服务器/计算资源的成本

通过投入冗余设备，我们可以进行常规的系统离线或其他预料之外的维护性操作。又或者可以利用一些空间来存储奇偶校验码块，以此来提供一定程度的数据持久性保证。

机会成本

这类成本由某一个组织承担。当该组织分配工程资源来构建减少风险的系统或功能，而非那些用户直接可用的功能时需要承担这些成本。这些工程师不能再从事为终端用户设计新功能和新产品的工作。

在SRE团队中，我们管理服务的可靠性很大程度上是通过管理风险来进行的。我们是将风险作为一个连续体来认知的。对于提高Google系统的可靠性和对服务故障的耐受水平，我们要给予同等关注。这样我们可以进行成本/收益分析。例如，Search、Ads、Gmail或者Photos应该置于风险连续体上（非线性）的哪一点？我们的目标是：明确地将运维风险与业务风险对应起来。我们会努力提高一项服务的可靠性，但不会超过该服务需要的可靠性。也就是说，当设定了一个可用性目标为99.99%时，我们即使要超过这个目标，也不会超过太多，否则会浪费为系统增加新功能、清理技术债务或者降低运营成本的机会。从某种意义上来说，我们把可用性目标同时看作风险的上限和下限。这种表达方式的主要优势在于它可以促使团队进行明确的、深思熟虑的风险讨论。

度量服务的风险

Google标准做法是通过一个客观的指标来体现一个待优化的系统属性。通过设立这样一个目标，我们可客观地评价目前的系统表现以及追踪一段时间内的改进和退步。对于服务风险而言，将所有的潜在因素缩减为一个单一的性能指标，可能不是一件能够立刻解决的事情。服务故

障可能会有很多潜在的影响，包括用户的不满、伤害，或丧失信任；直接或者间接的收入损失；品牌以及口碑上的影响；不良的新闻报道等。很明显，这些因素中的一部分很难被合理地度量。为了使这个问题在我们运行的各种类型的系统中易于处理，并且保持一致，我们选择主要关注计划外停机这个指标。

对于大多数服务而言，最直接的能够代表风险承受能力的指标就是对于计划外停机时间的可接受水平。计划外停机时间是由服务预期的可用性水平所体现的，通常我们愿意用提供的“9”系列的数字来体现，比如可用性为99.9%、99.99%或99.999%。每个额外的“9”都对应一个向100%可用性的数量级上的提高。对于服务系统而言，这个指标通常是基于系统正常运行时间比例的计算得出的（参考公式 3-1）。

公式**3-1**：基于时间的可用性

可用性＝系统正常运行时间/（系统正常运行时间＋停机时间）

使用这个公式，我们可以计算出一年内可接受的停机时间，从而可以使可用性达到预期目标。举例来说，一个可用性目标为99.99%的系统最多在一年中停机52.56分钟，就可以达到预计的可用性目标；详情参见附录A。

然而，在Google内部，基于时间的可用性通常是毫无意义的。因为我们需要着眼全球范围内的分布式服务。Google所采用的故障隔离手段使得我们能够保证在任何时候、任何地方对于一个给定的服务，总是可以处理一定的用户流量。（也就是说，我们随时都是部分“在线”的）。因此，我们通过请求成功率来定义服务可用性。公式3-2体现了这个基于产量的指标是怎样通过滚动窗口计算出来的（比如，一天内成功请求的比率）。

公式3-2：合计可用性

可用性＝成功请求数/总的请求数

例如，一个每天可用性目标为99.99%的系统，一天要接受2.5M个请求。它每天出现少于250个错误即可达到预计的可用性目标。

在一个典型的应用中，不是所有的请求都是平等的：一个新的用户注册请求失败和一个后台调用的新邮件的轮询请求失败是不同的。然而在许多情况下，从终端用户的角度来看，通过计算全部请求成功率是一个对于计划外停机时间的合理估计。

使用请求成功率指标量化计划外停机时间使得这种指标更适合在不直接服务终端用户的系统中使用。大多数非服务性的系统（比如，批处理、流水线、存储服务以及交易系统等）对成功和非成功的工作单元有明确的定义。事实上，虽然在本章中讨论的系统基本上都是有关消费者类型和基础设施服务类型的系统，但是同样的原则也基本上适用于非服务型系统。

例如，一个批处理进程通过提取、转换，并将客户数据库中的一位客户的数据内容插入数据仓库中，以便进行定期的进一步分析。通过使用某条记录处理成功和处理不成功来定义请求成功率，我们能计算出一个有效的可用性指标，尽管事实上该批处理系统并不是持续运行的。

通常，我们会为一项服务设定季度性的可用性目标，每周甚至每天对性能进行跟踪。我们通过寻找、跟踪和调整重要的、不可避免的偏差来使服务达到一个高层次的可用性目标。详情参见第4章。

服务的风险容忍度

如何辨别服务的风险容忍度？在一个正式的环境或安全关键的系统中，服务的风险容忍度通常是直接根据基本产品或服务的定义建立的。在 Google内部，服务风险容忍度往往定义得没有那么清楚。

为了辨别服务的风险容忍度，SRE必须与产品负责人一起努力，将一组商业目标转化为明确的可以实现的工程目标。这些商业目标会直接影响所提供服务的性能和可靠性目标。在实践中，这种转化说起来比做起来容易得多。消费者类型的服务往往有明确的产品负责人，而对于基础设施服务来说，拥有类似的产品所有权结构是很少见的（例如，存储系统或者通用的HTTP缓存层）。接下来，我们会分别讨论消费者服务和基础设施服务。

辨别消费者服务的风险容忍度

我们的消费者服务通常会有一个对应的产品团队，是该服务的商业所有者。比如说，Search、Google Maps和Google Docs，它们每一个都有自己的产品经理。这些产品经理负责了解用户和业务，在市场上塑造产品的定位。存在产品团队时，我们能够更好地通过这个团队来讨论服务的可靠性要求。在没有专门的产品团队的情况下，建立系统的工程师们经常在知情或不知情的情况下扮演了这个角色。

评价服务风险容忍度时，有许多需要考虑的因素。如下所示：

- 需要的可用性水平是什么？
- 不同类型的失败对服务有不同的影响吗？
- 我们如何使用服务成本来帮助在风险曲线上定位这个服务？
- 有哪些其他重要的服务指标需要考虑？

可用性目标

对于某个Google服务而言，服务的可用性目标通常取决于它提供的功能，以及这项服务在市场上是如何定位的。下面列出了要考虑的一些问题：

- 用户期望的服务水平是什么？
- 这项服务是否直接关系到收入（我们的收入或我们的客户的收入）？
- 这是一个有偿服务，还是免费服务？
- 如果市场上有竞争对手，那些竞争对手提供的服务水平如何？
- 这项服务是针对消费者还是企业的？

例如 Google Apps for Work，这个服务的主要用户是企业类用户，包括大型企业和中小企业。这些企业依靠Google应用程序提供的办公类

型的服务（例如Gmail、Calendar、Drive和Docs）让员工进行日常工作。另一方面，一个Google Apps for Work服务的中断不仅会影响Google本身，也会影响到那些在业务上非常依赖于我们的企业。对于这类服务，我们可能会设置季度性的外部可用性目标为99.9%。同时，我们会设置一个更高的内部可用性目标，以及签署一份如果我们未能达到外部目标的处罚性协议。

YouTube则需要截然不同的考虑。当Google收购YouTube后，我们需要为该网站提供一个更恰当的可用性目标。2006年，YouTube比当时的Google更加专注于消费者，并且当时处于一个与Google 非常不同的企业生命周期阶段。尽管当时YouTube已经有了一个很出色的产品，但它仍然在不断变化和快速发展着。因此，我们为YouTube设定了一个相比我们企业的产品更低的可用性目标，因为快速发展更加重要。

故障的类型

对于一项给定的服务的故障预期是另一个需要重点考虑的因素。我们的业务对于服务的停机时间的容忍程度有多高？持续的低故障率或者偶尔发生的全网中断哪一个会更糟糕？这两种类型的故障可能会导致绝对数量上完全相同的错误被返回，但可能对于业务的影响相差很大。

下面这个例子说明了一个提供私人信息的系统中自然发生的完全和部分服务中断的区别。假设有一个联系人管理应用程序，一种情况是导致用户头像显示失败的间歇性故障，另一种情况是将A用户的私人联系列表显示给B用户的故障。第一种情况显然是一个糟糕的用户体验，SRE会努力去快速地解决这个问题。然而，在第二种情况下，暴露私人数据的风险可能会破坏基本的用户信任。因此，在第二种情况下，在进行调试和事后的数据清理时，完全停止该服务更加恰当。

对于Google提供的其他服务，有时候，我们可以接受计划内的常规的服务中断。几年前，Ads前端曾经就是这样的一种服务。它是广告商和网站创建者用来建立、配置、运行和监控他们的广告活动的服务。因为这项工作大部分发生在正常工作时间内，我们认为维修窗口中发生的偶然的、正常的、计划之中的故障是可以接受的，并且我们把这些故障看作计划内停机时间，而不是计划外停机时间。

成本

决定一项服务的合理可用性目标时，成本是很重要的考虑因素。

广告服务就能很好地体现出这种取舍，因为成功与失败直接通过赢利和亏损体现。在为每一项服务确定可用性目标时，可以考虑如下的问题：

- 构建和运维可用性再多一个“9”的系统，收益会增加多少？

- 额外的收入是否能够抵消为了达到这一可靠性水平所付出的成本？为了使这个权衡等式更精确，假设一项业务中每一个请求的价值是一样的，考虑如下的成本与收益：

可用性目标：99.9%→99.99%

增加的可用性：0.09%

服务收入：100万美元

改进可用性后的价值：100万美元×0.0009＝900美元

在这种情况下，如果可用性提高一个“9”的成本不到900美元，这就是合理的投资。但是，如果成本超过900美元，那么成本将超过预计增加的收入。

当我们无法简单地解释可靠性和收入的关系时，可能会更难设置这些目标。这时，考虑在互联网中ISP的背景误差率可能是个不错的实用策略。如果从最终用户的角度测算故障，那么让服务的误差率低于背景误差率就是可能的。这些误差将归入给定用户的互联网链路的噪声当中。虽然ISP和各种应用协议之间差距很大（比如，TCP vs.UDP、IPv4 vs.IPv6），但是我们测算的ISP的背景误差率基本在0.01%和1%之间。

其他服务指标

除了可用性，其他指标也可以用来确定服务的风险容忍度。了解哪些指标是重要的，哪些指标是不重要的，可以为我们在讨论风险承受能

力时提供帮助。

Google广告系统的服务延迟是一个典型的例子。当Google发布Web搜索服务时，服务的一个很关键的特点就是速度快。AdWords是将广告显示在搜索结果旁边的系统。该系统的一个关键要求就是广告不能拖慢用户的搜索体验。这是每一代AdWords系统的一个不变的目标。

AdSense，允许出版商将JavaScript代码插入到自己的网站中的上下文广告，有一个非常不同的延迟目标。对于AdSense而言，延迟的目标是避免在插入上下文广告时影响到第三方页面。这个特定的延迟目标依赖于给定的发布者的页面呈现速度。这意味着AdSense的广告一般可以比AdWords的广告慢数百毫秒。

这种宽松的服务延迟要求允许我们在构建系统时做出一些明智的权衡（例如，使用的服务资源的数量和位置），这能够帮我们节省大量成本。换句话说，相对延迟不敏感的AdSense服务，使得我们能够通过合并需求较少的地理区域来降低运营开销。

基础设施服务的风险容忍度

构建和运维基础设施组件的要求在许多方面是不同于消费者服务的。一个根本的区别是，基础设施组件有多个客户，而他们通常有很多不同的需求。

可用性目标水平

Bigtable是一个大型结构化数据分布式存储系统，一些面向消费者的服务数据直接通过它服务用户请求。这样的服务需要很低的延迟和较高的可靠性。其他团队则把Bigtable作为离线分析的数据存储使用（例如，MapReduce）。这些团队往往更关注吞吐量而非可靠性。这两个情况的风险容忍度相当不同。

能够同时满足两种情况的要求的一种方法就是将所有基础设施服务做得极为可靠。但在实际情况中，这些基础设施服务往往需要占用大量资源，超高可靠性的代价通常是非常昂贵的。为了更好地了解不同类型用户的不同需求，我们可以分析每个用户对自己Bigtable请求队列的期

望。

故障类型

低延迟的用户希望Bigtable的请求队列（几乎总是）为空，这样系统可以立刻处理每个出现的请求。（事实上，效率低下的排队过程往往是导致较高的长尾延迟的原因。）而离线分析的用户更感兴趣的是系统的吞吐量，因此用户希望请求队列永远不为空。为了优化吞吐量，Bigtable系统应该避免处于空闲状态。

这样看来，对于这些用户而言，成功和失败是相反的。低延迟用户的成功是关注离线分析的用户的失败。

成本

一种在符合成本效益条件下满足这些竞争性约束的方式就是将基础设施分割成多个服务，在多个独立的服务水平上提供该服务。

在Bigtable的例子中，我们可以构建两个集群：低延迟集群和高吞吐量集群。低延迟集群是为那些需要延迟较低和可靠性较高的服务而设计的。为了确保队列长度最小和满足更严格的客户端隔离要求，Bigtable系统可以配备更多的冗余资源。另一方面，高吞吐量集群的配置冗余度较低，利用率较高。在实践中，高吞吐量集群只有低延迟集群成本的10%～50%。由于Bigtable是广泛部署的系统，这种成本上的降低非常显著。

基础设施服务运维的关键战略就是明确划分服务水平，从而使客户在构建系统时能够进行正确的风险和成本权衡。通过明确划定的服务水平，基础设施提供者其实就是将服务的成本的一部分转移给了用户。以这种方式暴露成本可以促使客户选择既能够满足他们的需求又能够压缩成本的服务水平。例如，Google＋将与保护用户隐私相关的数据放置在一个高可用、全球统一的数据存储中（例如，一个全球复制式的类似于SQL 的系统，Spanner，参见文献[cor12]），可选数据（不重要的但是能够增加用户体验的数据）放在一个价格更低、可靠性更低和最终一致的数据存储中（例如，Bigtable这种仅仅提供“尽力而为”模式的复制模

式的NoSQL存储系统）。

这里要注意的是，我们可以使用相同的硬件和软件运行多个级别的服务。可以通过调整服务的各种特性提供不同的服务水平，如资源的数量、冗余程度、资源的地理配置，以及基础设施软件的配置。

例子：前端基础设施

为了解释上文中介绍的这些风险容忍度评估原则不仅仅适用于存储基础设施，我们再来看一下另一大类型的服务：Google 的前端基础设施。这个前端基础设施是由反向代理和运行临近我们网络边缘的负载均衡系统组成的。

这些系统的核心工作是负责直接处理用户连接（例如，接受用户浏览器发出的TCP连接）。由于它们的关键性，这些系统需要具有超高的可靠性，面向消费者的服务通常可以用某种方式掩盖后端的不可用情况，但是这些基础服务一般没法这么做。如果某个请求没有到达应用服务的前端服务器，那么就意味着这个请求完全丢失了。

我们已经探讨了识别消费者服务和基础设施服务风险耐受能力的方法。现在，我们将继续讨论如何运用已知的风险耐受水平来通过错误预算调整系统的不可靠性。

使用错误预算的目的[1]

作者：**Mark Roth**

编辑：**Carmela Quinito**

本书的其他章节讨论了紧张局势之所以在产品研发小组和SRE 小组中产生，是因为他们基于不同的指标进行自己的绩效评估。产品研发的绩效是如何很大程度通过产品研发速度体现的，这会激励员工尽可能快地创建新的代码。同时，SRE 的绩效表现取决于该服务的可靠性，这意味着SRE 会对高频率的更改提出抗议。两个团队之间的信息不对称进一步加剧了这种内在的紧张局势。产品开发者更了解编写和发布他们的代码所需的时间，而SRE则更关注服务可靠性程度（以及生产环境中的其

他相关事项）。

这些紧张关系往往反映出他们自己对于应该投入到工程实践中的努力程度的不同意见。下面列举一些典型的紧张局势。

软件对故障的容忍度

对意外事件的容忍程度有多高？做得太少，我们就只能设计出一个脆弱无用的产品。

做得太多，我们的产品可能没有人会使用（但运行非常稳定）。

测试

同时，没有足够的测试，可能会有令人尴尬的故障发生，泄露隐私数据，或发生其他会上报纸的故障事件。而过于强调测试可能会失去市场先机。

发布频率

每一次发布都是有风险的。我们应该花多少时间在减少风险上？

金丝雀测试（Canary）的持续时间和大小

测试新发布的代码的最好做法就是在一个典型工作负载的服务子集中进行测试，这种做法通常被称为金丝雀测试。我们要等多久，而且测试范围有多大？

通常，现有团队已经找到了某种非正式的风险与成本的平衡点。不幸的是，人们很少能证明这种平衡是最佳的，而不是单单由参与谈判的工程师的谈判能力决定的。这些决定也不应该受办公室政治、不理智的恐惧或一厢情愿的希望所驱使（事实上，Google SRE的非官方座右铭就是“希望不是一种策略”）。

相反，我们的目标是定义一个双方都同意的客观指标，该指标可以用来指导谈判的方向。一项决策越是基于数据做出的，常常就越好。

错误预算的构建过程

为了基于客观数据做出决策，两个团队需要共同定义一个基于服务水平目标（SLO）的季度错误预算（参考第4章）。错误预算提供了一个明确的、客观的指标来决定服务在一个单独的季度中能接受多少不可靠性。这个指标在SRE与产品研发部门的谈判中将政治因素排除。

我们的实际做法如下：

- 产品管理层定义一个SLO，确定一项服务在每个季度预计的正常运行时间。

- 实际在线时间是通过一个中立的第三方来测算的：我们的监控系统。

- 这两个数字的差值就是这个季度中剩余的不可靠性预算。

- 只要测算出的正常在线时间高于SLO，也就是说，只要仍然有剩余的错误预算，就可以发布新的版本。

例如，假设一项服务的SLO是每季度99.999%的查询成功率。这就意味着，这项服务在给定季度的错误预算是0.001%。如果某个问题导致我们这个季度产生了0.0002%的故障，就相当于占用了这项服务20% 的季度错误预算。

好处

错误预算的主要好处就是它能够激励产品研发和SRE一起找出创新和可靠性之间合理的平衡点。

许多产品使用这个控制回路来调节发布的速度：只要系统符合SLO，就可以继续发行新版本。如果频繁地违反SLO 导致错误预算被耗尽，那么发布就会暂停，同时需要在系统测试和开发环节投入更多资源使得系统更有弹性，以使性能得到提升。

有比这种简单的开/关技术更巧妙和有效的方法：[2]例如，当SLO违规导致错误预算接近耗尽时，将发布的速度减慢，或者回退到上一版本。

例如，如果产品研发人员想要在测试上节约时间或者想要提高发布速度并且SRE 表示反对时，那么就可以通过错误预算指导决策。当预算剩余很多时，产品研发人员就可以承担更多的风险。如果预算接近耗尽，产品研发人员自身将会推动更多的测试或者放慢发布的速度，因为他们不想冒着用尽预算的风险和拖延他们的程序上线。实际上，产品开发团队这样就开始进行自我监管。他们知道预算还剩多少，并且可以控制自己的风险。（当然，这要求SRE在SLO达不到的时候有权停止程序的发布。）

如果网络中断或者数据中心发生故障影响了SLO，怎么办？这样的事件也会给错误预算带来不良的影响，会使本季度剩余部分的发布将会减少。整个团队会支持这种发布频率的降低，因为每个人都有义务保障服务正常运行。

利用错误预算可以同时找到制定得过高的可用性目标，显示出它们所导致的灵活性和创新速度方面的问题。如果团队无法发布新的功能，他们可以选择降低SLO（从而增加错误预算）来提高创新速度。

关键点

- 管理服务的可靠性主要在于管理风险，而且管理风险的成本可能很高。

- 100%可能永远都不是一个正确的可靠性目标：不仅是不可能实现的，而且它通常比一项服务的用户期望的可靠性大得多。我们要将服务风险和愿意承担的业务风险相匹配。

- 错误预算在SRE和产品研发团队之间调整激励，同时强调共同责任。错误预算使得讨论发布速率更容易，同时可有效地减少任何关于事故的讨论。这样，多个团队可以毫无怨言地对生产环境风险度达成一致。

第4章 服务质量目标

作者：**Chris Jones、John Wilkes、Niall Murphy、Cody Smith**

编辑：**Betsy Beyer**

如果不详细了解服务中各种行为的重要程度，并且不去度量这些行为的正确性的话，就无法正确运维这个系统，更不要说可靠地运维了。那么，不管是对外服务，还是内部API，我们都需要制定一个针对用户的服务质量目标，并且努力去达到这个质量目标。

在这个过程中，我们需要利用一些主观判断结合过去的经验以及对服务的理解来定义一些服务质量指标（SLI）、服务质量目标（SLO），以及服务质量协议（SLA）。这三项分别是指该服务最重要的一些基础指标、这些指标的预期值，以及当指标不符合预期时的应对计划。事先选择好合适的指标有助于在故障发生时帮助SRE进行更好地决策，同时也为SRE团队判断系统是否正常工作提供帮助。

本章描述了SRE团队在指标建模、指标选择，以及指标分析上采用的基本框架。我们会采用前文介绍的莎士比亚搜索服务作为示例来展开讨论，具体详见第2章的“莎士比亚搜索服务”。

服务质量术语

很多读者可能都对SLA这个概念非常熟悉，但SLI与SLO则可能需要一个详细定义。因为在常见的使用环境中，SLA经常被赋予了过多的意义，Google则更倾向于利用不同的词语描述其中不同的涵义，以便更加清晰地进行讨论。

指标

SLI是指服务质量指标（indicator）—该服务的某项服务质量的一个具体量化指标。

大部分服务都将请求延迟—处理请求所消耗的时间—作为一个关键SLI。其他常见的SLI包括错误率（请求处理失败的百分比）、系统吞吐量（每秒请求数量）等。这些度量通常是汇总过的：在某一个度量时间范围内将原始数据收集起来，计算速率、平均值、百分比等汇总数据。

理想情况下，SLI应该直接度量某一个具体的服务质量。但是很多时候，直接度量信息可能非常难以获取，或者无法观测，我们只能用某种指标来替代。例如，客户端的延迟数据经常是最直接的用户指标，但是由于条件限制可能只能监控服务器端的延迟数据。

可用性（availability）是另外一个SRE重视的SLI，代表服务可用时间的百分比。该指标通常利用“格式正确的请求处理成功的比例”来定义，有时也称为服务产出（yield）。对数据存储系统来说，持久性（durability）—数据能够完整保存的时间—也是一个重要指标。虽然100%的“可用性”是不可能实现的，但是接近100%的可用性指标是可以实现的一个目标。运维行业经常用9的数量来描述可用程度。例如，99%可用性被称为“2个9”,99.999%被称为“5个9”。目前Google　云计算服务公开的可用性指标是“3.5个9”— 99.95% 可用。

目标

SLO是服务质量目标（Objective）：服务的某个SLI的目标值，或者目标范围。SLO的定义是SLI≤目标值，或者范围下限≤SLI≤范围上限。例如，对莎士比亚服务来说，返回结果的速度应该是很“快”的，那么我们可以定义一个SLO，要求搜索请求的平均延迟小于100ms。

选择一个合适的SLO是非常复杂的过程。第一个困难点是很有可能无法确定一个具体的值。例如对外部传入的HTTP请求来说，每秒查询数量（QPS）指标是由用户决定的，我们并不能针对这个指标设置一个SLO。

但是，我们可以做的是指定平均请求延迟小于100ms，确立这个目标可以鼓励开发者优化前端服务降低延迟，或者采购某种延迟更低的硬件。（100ms在这里只是一个随意选择的值，但是一般来说速度快要比速度慢更好，因为用户可见的延迟超过一定数量后会使得参与程度下降。详情参见“Speed Matter”，文献[Bru09]中有详细介绍）。

而且，可能不那么直观的是，这两个SLI—QPS和延迟—很可能是相关的：QPS升高通常会导致延迟升高，服务到达一定负载水平后性能下降是很常见的。

SLO的选择和公布可以帮助设立用户对服务质量的预期。该策略可以应对那些没有根据的抱怨—“服务太慢了”。如果没有一个明确的SLO，用户经常会按照自己的理解设置一个服务性能的预期，即使这可能跟运维人员或者设计者所想的完全不同。这种问题可能会导致对某个服务的过度依赖—用户错误地认为这个服务会比实际情况更可靠（例如Chubby的例子，参见下面的“全球Chubby服务计划内停机”。另外一种情况是会导致信心不足—用户会认为系统比实际情况更脆弱和不可靠，从而不会去使用它。

全球**Chubby**服务计划内停机

作者：**Marc Alvidrez**

Chubby是Google的一个分布式锁服务，用于松散耦合的分布式系统。在全球Chubby服务中，我们将Chubby实例的副本分布在不同的地理区域。随着时间的推移，我们发现，全球实例出现问题经常会导致其他服务出现故障，其中很多故障都会影响到外部用户。但是，由于真正的全球Chubby服务故障出现的频率太低，以至于其他服务负责人开始认为全球Chubby服务永远不会出故障，从而不停地将更多的服务依赖于此。Chubby全球服务的高可靠性实际上提供了一种安全假象，因为这些服务实际上在Chubby全球服务不可用的时候不能正常工作，不管这种情况是多么罕见。

我们在这里采用了一个很有趣的解决办法：SRE保证全球Chubby服务能够达到预定义的SLO，但是同时也会确保服务质量不会大幅超出该SLO。每个季度，如果真实故障没有将可用性指标降低到SLO之下，SRE会有意安排一次可控的故障，将服务停机。利用这种方法，我们可以很快找出那些对Chubby全球服务的不合理依赖，强迫服务的负责人尽早面对这类分布式系统的天生缺陷。

协议

最后，SLA是服务质量协议（Agreement）：指服务与用户之间的一个明确的，或者不明确的协议，描述了在达到或者没有达到SLO之后的后果。这些后果可以是财务方面的—退款或者罚款—也可能是其他类

型的。区别SLO和SLA的一个简单方法是问“如果SLO没有达到时，有什么后果？”如果没有定义明确的后果，那么我们就肯定是在讨论一个SLO，而不是SLA。[3]

SRE通常不会参与SLA的书写，因为SLA是与业务产品的决策紧密相关的。但是，SRE确实会参与帮助避免触发SLA中的惩罚性条款。同时，SRE会参与制定具体的SLI：很明显，提供一个客观的方式来度量SLO是很重要的，否则大家就会产生分歧。

Google搜索服务是没有公开SLA的一个典型服务：我们当然希望所有人都能够最方便、最快地使用Google的搜索服务，但是我们并没有与全世界签订合同。但是，即使如此，如果搜索服务不可用依然有后果产生—对Google形象有损害，同时也会使得广告业务收入下降。很多其他的Google服务，例如Google for Work，具有明确的用户SLA。不管某个服务是否具有SLA，定义SLI与SLO，并且用它们来管理服务质量都是很有价值的。

理论说了这么多，终于可以开始讲一些实践经验了。

指标在实践中的应用

既然我们已经详细描述了为什么选择合适的指标度量服务质量是很重要的，那么究竟如何来识别哪些指标对服务是最重要的呢？

运维人员和最终用户各关心什么

我们不应该将监控系统中的所有指标都定义为SLI；只有理解用户对系统的真实需求才能真正决定哪些指标是否有用。指标过多会影响对那些真正重要的指标的关注，而选择指标过少则会导致某些重要的系统行为被忽略。一般来说，四五个具有代表性的指标对系统健康程度的评估和关注就足够了。

常见的服务，根据它们的相关SLI通常会归类为以下几个大类。

- 用户可见的服务系统，例如莎士比亚搜索服务的前端服务器通常关心可用性、延迟，以及吞吐量。换句话说：是否能正常处理请求？每

个请求花费的时间是多少？多少请求可以被处理？

- 存储系统通常强调：延迟、可用性和数据持久性。换句话说：读写数据需要多少时间？我们是否可以随时访问数据？数据是否一段时间内还能被读取？扩展讨论参见第26章。

- 大数据系统，例如数据处理流水线系统，一般来说关心吞吐量和端到端延迟。换句话说：处理了多少数据？数据从输入到产出需要多少时间？（某些流水线任务还会关注某个单独处理阶段的延迟。）

- 所有的系统都应该关注：正确性。是否返回了正确的回复，是否读取了正确的数据，或者进行了正确的数据分析操作。正确性是系统健康程度的一个重要指标，但是它更关注系统内部的数据，而不是系统本身，所以这通常不是SRE直接负责的。

指标的收集

利用某种监控系统，大部分指标数据都在服务器端被收集，例如Borgmon（具体参见第10章）或者Prometheus。或者利用某种日志分析系统，例如分析日志中HTTP 500回复所占的比例。然而，某些系统可以加入对客户端数据的收集，否则可能会错失一些不影响服务器端指标，但是对用户产生影响的问题。例如，只关注莎士比亚服务器搜索后端的延迟可能会错失由页面JavaScript脚本导致的用户可见的延迟问题。在这个例子中，度量页面在浏览器中可用的延迟是度量用户体验的一个更好的指标。

汇总

为了简化和使数据更可用，我们经常需要汇总原始度量数据。汇总过程应该非常小心。

某些指标的汇总看起来是很简单的，例如每秒服务请求的数量，但是即使这种简单度量也需要在某个度量时间范围内进行汇总。该度量值是应该每秒获取一次，还是每分钟内的平均值？后者可能会掩盖仅仅持续几秒的一次请求峰值。假设某个系统在偶数秒处理200个请求，在其他时间请求为0。该服务与持续每秒处理100个请求的服务平均负载是一

样的，但是在即时负载上却是两倍。同样的，平均请求延迟可能看起来很简单，但是却掩盖了一个重要的细节；很可能大部分请求都是很快的，但是长尾请求速度却很慢。

大部分指标都应该以“分布”，而不是平均值来定义。例如，针对某个延迟SLI，某些请求可能很快，其他的可能会很慢，有时候会非常慢。简单平均可能会掩盖长尾延迟和其中的变化。图4-1提供了一个例子：虽然常见请求可以在50ms完成，但5%的请求却慢了20倍！针对平均值的监控和报警将不会发现任何改变，但是服务的确在长尾延迟上出现了巨大变化（图中最上面的那条线）。

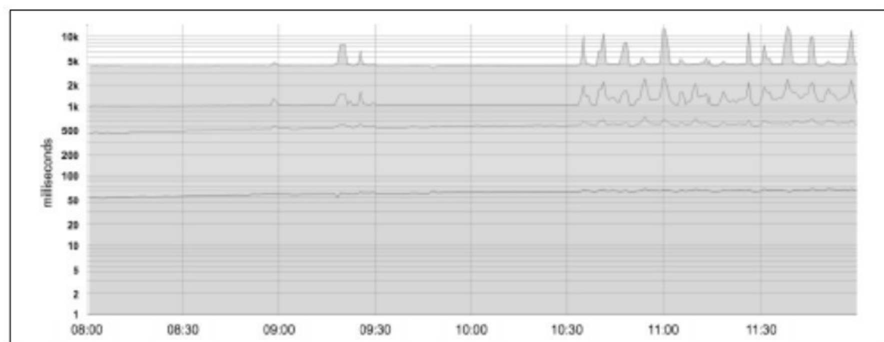


图4-1：某系统的 50%、85%、95%、99%的请求延迟，注意这里的Y轴是指数级分布的。

利用百分位指标可以帮助我们关注该指标的分布性：高百分位，例如99% 和99.9% 体现了指标的最差情况，而50% 则体现了普遍情况（99%百分位是指在原始数据中99%的数值都满足某种条件。例如请求延迟的99%为100ms指的是，在所有请求中，99%的请求延迟都小于100ms）。响应时间的分布越分散，意味着普通用户受到长尾请求延迟的影响就越明显，这可能预示了负载过高情况下出现的排队问题。用户研究显示，用户通常更喜欢速度较慢的系统，而不是一个请求速度抖动很厉害的系统，所以，某些SRE团队只关注长尾部分，因为如果99.9%的系统行为都正常的话，那50%部分就肯定也是正常的。

关于统计性谬误

一般来说，SRE更倾向于分析一组数据的百分比分布，而非其算术平均值。长尾效应比算术平均值更有特点，使用百分比分布能够更清晰

地进行分析。因为计算机系统的本身特质决定，数据是具有特定分布特点的 ——例如，请求延迟必须大于0，同时如果超时设置为1000ms，则不可能有成功请求超过这个时间。因此，我们不能假设算术平均值和中位数是相等的——它们甚至可能相差甚远！

我们同时也不会假设数据是平均分配的，一切都必须经过验证。某些常见的直觉感觉和近似方法在这里并不适用。例如，如果分布情况和假设不一致，那么根据这个假设做出的操作就有可能频率过高，或者频率过低（例如根据请求延迟将过高的服务器自动重启）。

指标的标准化

我们建议标准化一些常见的SLI，以避免每次都要重新评估它们。任何一个符合标准定义模板的服务可以不需要再次自己定义SLI。

- 汇总间隔：每1分钟汇总一次
- 汇总范围：集群中的全部任务
- 度量频率：每10秒一次
- 包含哪些请求：从黑盒监控任务发来的HTTP GET请求
- 数据如何获取：通过监控系统获取服务器端信息得到
- 数据访问延迟：从收到请求到最后一个字节被发出

为了节约成本，应该为常见的指标构建一套可以重用的SLI模板，从而使得理解每个SLI更简单。

目标在实践中的应用

我们应该从思考（或者调研）用户最关心的方面入手，而非从现在能度量什么入手。用户真正关心的部分经常是度量起来很困难，甚至是不可能的，所以我们需要以某种形式近似。然而，如果我们只是从可以简单度量的数值入手，最终的SLO的作用就会很有限。因此，与其选择指标，再想出对应的目标，不如从想要的目标反向推导出具体的指标。

目标的定义

为了更清晰地定义，SLO应该具体指出它们是如何被度量的，以及其有效条件。例如，我们可能说：

- 99%（在1分钟的时间范围内汇总）的Get RPC调用会在小于100ms的时间内完成（包括全部后端服务器）。

- 99% 的Get RPC会在100ms内完成（这一句与上一句一样，只是利用了SLI模板中的信息减少了重复信息）。

如果性能曲线也很重要的话，我们可以指定多个SLO目标：

- 90% 的Get RPC会在1ms内完成。

- 99% 的Get RPC 会在10ms内完成。

- 99.9% 的Get RPC会在100ms内完成。

如果我们同时具有批处理用户（关注吞吐量）以及在线交互式用户（关注延迟），那么可能应该为每种负载指定单独的SLO目标：

- 95% 的批处理用户Set RPC 应该在1s之内完成。

- 99% 的交互式用户Set RPC，并且RPC负载小于 1KB 的应该在10ms之内完成。

要求SLO能够被100% 满足是不正确，也是不现实的：过于强调这个会降低创新和部署的速度，增加一些成本过高、过于保守的方案。更好的方案是使用错误预算（Error Budget）——对达不到SLO的容忍度——以天或者以周为单位计量。高层管理者可能同时也需要按月度或者季度的评估。（错误预算其实就是保证达到其他SLO的一个SLO!）

达不到SLO的现象的发生频率对用户可见的服务健康度来说是一个有用的指标。通过每日（或者每周）对SLO达标程度的监控可以展示一个趋势，这样就可以在重大问题发生之前得到预警。

SLO不达标的频率可以用来与错误预算进行对比（见第3章“使用错

误预算的目的”一节），利用这两个数值的差值可以指导新版本的发布。

目标的选择

选择目标SLO不是一个纯粹的技术活动，因为这里还涉及产品和业务层面的决策，SLI和SLO（甚至SLA）的选择都应该直接反映该决策。同样的，有时候可能可以牺牲某些产品特性，以便满足人员、上线时间（time to market）、硬件可用性，以及资金的限制。SRE应该积极参与这类讨论，提供有关可行性和风险性的建议，下面列出了一些有用的讨论。

不要仅以目前的状态为基础选择目标

了解系统的各项指标和限制非常重要，但是仅仅按照当前系统的标准制定目标，而不从全局出发，可能会导致团队被迫长期运维一个过时的系统，没有时间去推动架构重构等任务。

保持简单

SLI中过于复杂的汇总模式可能会掩盖某种系统性能的变化，同时也更难以理解。

避免绝对值

虽然要求系统可以在没有任何延迟增长的情况下无限扩张，或者“永远”可用是很诱人的，但是这样的要求是不切实际的。就算有一个系统能够做到这一点，它也需要花很长时间来设计和构建，同时运维也很复杂—最关键的是，这可能比用户可以接受的（甚至是很开心地接受的）标准要高太多。

SLO越少越好

应该仅仅选择足够的SLO来覆盖系统属性，一定要确保每一个SLO都是必不可少的：如果我们无法针对某个SLO达标问题说服开发团队，那么可能这个SLO就是不必要的[4]。然而，不是所有的产品属性都能用SLO表达，用户的“满意度”就很难。

不要追求完美

我们可以随着时间流逝了解系统行为之后优化SLO的定义。刚开始可以以一个松散的目标开始，逐渐收紧。这比一开始制定一个困难的目标，在出现问题时放松要好得多。

SLO 可以成为—也应该成为—SRE和产品团队划分工作优先级的重要参考，因为SLO代表了用户体验的程度。好的SLO是对开发团队有效的、可行的激励机制。但是一个没有经过精心调校的SLO会导致浪费，某团队可能需要付出很大代价来维护一个过于激进的SLO，而如果SLO过于松散，则会导致产品效果很差。SLO是一个很重要的杠杆：要小心使用。

控制手段

SLI和SLO在决策系统运维时也非常有用：

1.监控并且度量系统的SLI。

2.比较SLI和SLO，以决定是否需要执行操作。

3.如果需要执行操作，则要决定究竟什么操作需要被执行，以便满足目标。

4.执行这些操作。

例如，如果在第2步中，请求延迟正在上涨，无操作的话，会在几个小时内超出SLO范围。第3步则会测试服务器是否是CPU资源不够，同时增加一些CPU来分散负载。没有SLO的话，我们就不知道是否（或者何时）需要执行该动作。

SLO可以建立用户预期

通过公布SLO可以设置用户对系统行为的预期。用户（以及潜在用户）经常希望知道他们可以预期的服务质量，以便理解该服务是否能够满足他们的要求。例如，某个团队想要开发一个照片分享网站，可能会避免使用一个数据持久性高、成本低，但是可用性低的系统。但是，可

能会使用另外一个存档信息管理系统。

为了让用户拥有正确的预期，我们可以考虑使用以下几种策略：

留出一定的安全区

对内使用更高的SLO，对外使用稍低的SLO可以留出一些时间用来响应问题。SLO缓冲区也可以用来进行可能影响系统属性的重构，例如降低成本以及方便运维等，缓冲区保护我们不会对用户产生直接影响。

实际SLO也不要过高

用户一般不会严格按照书本定义的SLO，而是按照实际情况来构建服务，这在基础设施类服务上非常明显。如果服务的实际性能要比SLO宣传得好太多，用户可能会逐渐依赖于现在的假象。我们可以利用主观可控模式减少这种过度依赖。（Google Chubby服务会用计划内维护来避免过于可用。）[5]同时，可以采取对一些请求限速，或者限制系统在低负载情况下也不会速度过快的手段。

理解系统行为与预期的符合程度可以帮助决策是否需要投入力量优化系统，使其速度更快、更可用，或者更可靠。如果服务一切正常，可能力量应该花在其他的优先级上，例如消除技术债务、增加新功能，或者引入其他产品等。

协议在实践中的应用

起草一份SLA需要业务部门和法务部门选择合适的后果条款。SRE在这个过程中的作用是帮助这些部门理解SLA的SLO达标的概率和困难程度。许多针对SLO的建议也同样适用于SLA。最好在用户宣传方面保持保守，因为受众越广，修改和删除一个不合适或者很有困难达到的SLA就越困难。

第5章 减少琐事

作者：**Vivek Rau**

编辑：**Betsy Beyer**

如果系统正常运转中需要人工干预，应该将此视为一种Bug。

“正常”的定义会随系统的进步而不断改变。

Carla Geisser,Google SRE

SRE要把更多的时间花费在长期项目研发上而非日常运维中。因为术语日常运维可能会被误解，我们在这里使用一个专门的词语——琐事（toil）。

琐事的定义

琐事不仅仅代表“我不喜欢做的工作”。它也不能简单地等同于行政杂务加上其他脏活累活。每个人满意和喜欢的工作类型是不同的，有的人很喜欢手工的、重复性的工作。同时，一些管理类杂务是必须做的，不应该被归类为琐事：这些是流程开销（overhead）。流程开销通常是指那些和运维产品服务不直接相关的工作，包括团队会议、目标的建立和评估[6]、每周总结[7]以及人力资源的书面工作等。而脏活累活通常具有长期价值，这些也不能算作琐事。例如，为服务清理警报规则或降低噪声率可能是一件繁重的工作，但这些不是琐事。

到底什么是琐事？琐事就是运维服务中手动性的，重复性的，可以被自动化的，战术性，没有持久价值的工作。而且，琐事与服务呈线性关系的增长。并不是每件琐事都有以上全部特性，但是，每件琐事都满足下列一个或多个属性：

手动性

例如手动运行脚本以便自动执行一些任务。运行一个脚本可能比手动执行脚本中的每一步要快，但具体运行脚本所花费的手动的时间（而非脚本所需要的运行时间）应该被认为是琐事。

重复性的

如果某件事是第一次做，甚至第二次做，都不应该算作琐事。琐事就是不停反复做的工作。如果你正在解决一个新出现的问题或者寻求一种新的解决办法，不算作琐事。

可以被自动化的

如果计算机可以和人类一样能够很好地完成某个任务，或者通过某种设计变更来彻底消除对某项任务的需求，这项任务就是琐事。如果主观判断是必需的，那么很大程度上这项任务不属于琐事。[8]

战术性的

琐事是突然出现的、应对式的工作，而非策略驱动和主动安排的。处理紧急警报是琐事。我们可能永远无法完全消除这种类型的工作，但我们必须继续努力减少它。

没有持久价值

如果在你完成某项任务之后，服务状态没有改变，这项任务就很可能是琐事。如果这项任务会给服务带来永久性的改进，它就不是琐事。一些繁重的工作—比如挖掘遗留代码和配置并且将它们清理出去也不是琐事。

与服务同步线性增长

如果在工作中所涉及的任务与服务的大小、流量或用户数量呈线性增长关系，那这项任务可能属于琐事。一个良好管理和设计的服务应该至少可以应对一个数量级的增长，而不需要某些一次性工作（例如增加资源）之外的额外工作。

为什么琐事越少越好

SRE的一个公开目标是保持每个SRE的工作时间中运维工作（即琐事）的比例低于50%。SRE至少花50%的时间在工程项目上，以减少未来的琐事或增加服务功能。增加服务功能包括提高可靠性、性能，或利用率，同时也会进一步消除琐事。

SRE公开50%这个目标是因为如果不加以控制，琐事会变得越来越多，以至于迅速占据我们每个人100%的时间。减少琐事和扩大服务规模的工作就是 SRE 中的E（Engineering）。对工程工作的关注使SRE可以在服务规模扩大的同时减少人数，并且比单纯的研发团队和单纯的运维工作团队能更有效地管理服务的秘诀。

不仅如此，招聘新的SRE时，我们也会引用上文提及的50%规则，承诺新员工不会专门进行运维工作。我们通过禁止SRE组织或者其中任何小团队退化为专门从事运维工作的组织来实现这个承诺。

琐事的计算

如果我们想要将一个SRE花在琐事上的时间限制在50%，应该如何分配时间呢？

任何一个SRE在参与on-call时都会承担一定程度的琐事。一个典型的SRE每个周期中会有一周主on-call和一周副on-call的工作。（主on-call和副on-call的讨论，请参见第11 章）。因此，在一个六个人的轮值周期中，每六周中至少有两周需要专注于on-call和中断性事务的处理，这意味着潜在的琐事的最小值是一个SRE的工作时间的2/6，也就是33%。如果是八人轮值，那么最小值就是2/8，即25%。

与此计算相一致，来自SRE的数据显示，琐事的最大来源就是中断性工作（即与服务相关的非紧急的邮件和电子邮件）。另一个主要来源是on-call（紧急的），紧随其后的是发布和数据更新。即使Google的发布和数据更新过程通常是高度自动化的，这个部分仍有许多改进的空间。

Google SRE的季度调查显示，琐事的时间占用大约在33%，所以我们其实目前比总体目标 50% 做得更好。然而，这个平均值没有显示出其中的异常情况：一些SRE 琐事比例为0%（纯开发项目而不参加on-call），而其他人宣称他们在琐事上花费了80% 的时间。当某个SRE报告自己承担了过量的琐事时，这通常意味着管理者需要在团队中更均衡地分布琐事负荷，同时应该鼓励该SRE找到自己满意的工程项目。

什么算作工程工作

工程工作（Engineering）是一种新颖的、本质上需要主观判断的工作。它是符合长期战略的，会对你的服务进行长久性的改善的工作。工程工作通常是有创新性和创造性的，着重通过设计来解决问题，解决方案越通用越好。工程工作有助于使该团队或是整个SRE组织在维持同等人员配备的情况下接手更大或者更多的服务。

典型的SRE活动分为如下几类。

软件工程

编写或修改代码，以及所有其他相关的设计和文档工作。例如，编写自动化脚本，创造工具或框架，增加可扩展性和可靠性的服务功能，或修改基础设施代码以使其更稳健。

系统工程

配置生产系统、修改现存配置，或者用一种通过一次性工作产生持久的改进的方法来书写系统文档。例如，监控的部署和更新、负载均衡的配置、服务器配置、操作系统的参数调整和负载均衡器的部署。系统工程还包括与研发团队进行的架构、设计和生产环境方面的咨询工作。

琐事

与运维服务相关的重复性的、手工的劳动。

流程负担

与运维服务不直接相关的行政工作。例如招聘、人力资源书面工作、团队/公司会议、任务系统的定期清理工作、工作总结、同行评价和自我评价，以及培训课程等。

按全年或者数个季度来说，每个SRE需要花费至少50%的时间在工程工作中。琐事通常有一定集中性，对于某些团队而言，把50%的时间稳定地花在工程工作上可能不太现实，在某些季度中可能无法达到这个目标。但是，长期看来，如果工程时间的比例大幅低于50%，受影响的

团队就需要退一步来找出问题所在。

琐事繁多是不是一定不好

琐事不会总是让每个人都不开心，特别是不太多的时候。已知的和重复性的工作有一种让人平静的功效。完成这些事可以带来一种满足感和快速胜利感。琐事可能是低风险低压力的活动，有些员工甚至喜欢做这种类型的工作。

琐事的存在并不总是坏事，但是每个人都必须清楚，在SRE所扮演的角色中，一定数量的琐事是不可避免的，这其实是任何工程类工作都具有的特点。少量的琐事存在不是什么大问题。但是一旦琐事的数量变多，就会有害了。如果琐事特别繁重，那就应该非常担忧，大声抱怨。在许多琐事有害的原因中，有如下因素需要考虑：

职业停滞

如果花在工程项目上的时间太少，你的职业发展会变慢，甚至停滞。Google确实会奖励做那些脏活累活的人，但是仅仅是该工作是不可避免，并有巨大的正面影响的时候才会这样做。没有人可以通过不停地做脏活累活满足自己的职业发展。

士气低落

每个人对自己可以承担的琐事限度有所不同，但是一定有个限度。过多的琐事会导致过度劳累、厌倦和不满。

另外，牺牲工程实践而做琐事会对SRE组织的整体发展造成损害，原因如下：

造成误解

我们努力确保每个SRE以及每个与SRE一起工作的人都理解SRE是一个工程组织。

如果个人或者团队过度参与琐事，会破坏这种角色，造成误解。

进展缓慢

琐事过多会导致团队生产力下降。如果SRE团队忙于为手工操作和导出数据救火，新功能的发布就会变慢。

开创先例

如果SRE过于愿意承担琐事，研发同事就更倾向于加入更多的琐事，有时候甚至将本来应该由研发团队承担的运维工作转给SRE来承担。其他团队也会开始指望SRE接受这样的工作，这显然是不好的。

促进摩擦产生

即使你个人对琐事没有怨言，你现在的或未来的队友可能会很不开心。如果团队中引入了太多的琐事，其实就是在鼓励团队里最好的工程师开始寻找其他地方提供的更有价值的工作。

违反承诺

那些为了项目工程工作而新入职的员工，以及转入SRE的老员工会有被欺骗的感觉，这非常不利于公司的士气。

小结

如果我们都致力于每一周通过工程工作消除一点琐事，就可以持续性地整顿服务。我们就可以将更多的力量投入到扩大服务规模的工程工作上去，或者是进行下一代的服务的架构设计，又或者是建立一套跨SRE使用的工具链。让我们多创新，少干琐事吧！

第6章 分布式系统的监控

作者：**Rob Ewaschuk**

编辑：**Betsy Beyer**

Google的SRE 团队在构建监控系统和报警系统方面遵循一些核心思

想和最佳实践。本章在决定何时需要人工干预（发出紧急警报）的问题上提供了一些指导意见，同时也讨论了如何应对那些不那么严重的警报。

术语定义

在讨论监控系统时，目前几乎没有通用的术语。即使在Google内部，不同的团队也在使用不同的术语，以下是绝大部分通用的术语。

监控（monitoring）

收集、处理、汇总，并且显示关于某个系统的实时量化数据，例如请求的数量和类型，错误的数量和类型，以及处理用时，应用服务器的存活时间等。

白盒监控（white-box monitoring）

依靠系统内部暴露的一些性能指标进行监控。包括日志分析、Java虚拟机提供的监控接口，或者一个列出内部统计数据的HTTP接口进行监控。

黑盒监控（black-box monitoring）

通过测试某种外部用户可见的系统行为进行监控。

监控台页面（dashboard）

提供某个服务核心指标一览服务的应用程序（一般是基于Web的）。该应用程序可能会提供过滤功能（filter）、选择功能（selector）等，但是最主要的功能是用来显示系统最重要的指标。该程序同时可以显示相应团队的一些信息，包括目前工单的数量、高优先级的Bug列表、目前的on-call工程师和最近进行的生产发布等。

警报（alert）

目标对象是某个人发向某个系统地址的一个通知。目的地可以包括工单系统、E-mail地址，或者某个传呼机。相应的，这些警报被分类

为：工单、E-mail警报[9]，以及紧急警报（page）。

根源问题（root cause）

指系统（软件或流程）中的某种缺陷。这个缺陷如果被修复，就可以保证这种问题不会再以同样的方式发生。某一个故障情况可能同时具有多个根源问题：例如，有可能自动化程度不够，软件在异常输入下崩溃，以及对生成配置文件的脚本测试不足等。这里每一个因素都是一个根源问题，并且每一个都需要被修复。

节点或者机器（node/machine）

这里的两个术语是可以互换的：指在物理机、虚拟机，或者容器内运行的某个实例。

某个单独的物理机器上可能有多个服务需要监控，这些服务可能具有如下特点。

- 相互关联的服务：例如Web服务器与对应的缓存服务器。
- 不相关的服务，它们仅仅共享硬件：例如代码仓库和把文件存放在代码仓库中的配置管理系统的主进程。例如Puppet（https://puppetlabs.com/puppet/puppetopen-source）和Chef（https://www.chef.io/chef/）。

推送（push）

关于某个服务正在运行的软件或者其配置文件的任何改动。

为什么要监控

监控一个系统有多个原因，包括如下几项。

分析长期趋势

数据库目前的数据量，以及增长速度。又例如每日活跃用户的数量增长的速度。

跨时间范围的比较，或者是观察实验组与控制组之间的区别

使用Acme Bucket of Bytes 2.72 或者Ajax DB 3.14（都是虚构的系统名称名字）哪个请求速度更快？增加新节点后，memcache的缓存命中率是否增加？网站是否比上周速度要慢？

报警某项东西出现故障了，需要有人立刻修复！或者某项东西可能很快会出故障，需要有人尽快查看。

构建监控台页面

监控台页面应该可以回答有关服务的一些基本问题，通常会包括常见的4个“黄金指标”（参见本章后面“4个黄金指标”一节中的详细讨论）。

临时性的回溯分析（也就是在线调试）

我们的请求延迟刚刚大幅增加了，有没有其他的现象同时发生？

系统监控在给业务分析提供原始数据和分析安全入侵的场景时也有一定作用。但是由于本书关注的焦点是SRE所关注的工程领域，我们就不在这里讨论这些方面的应用了。

监控与报警可以让一个系统在发生故障时主动通知我们，或者能够告诉我们即将发生什么。当系统无法自动修复某个问题时，需要一个人来调查这项警报，以决定目前是否存在真实故障，采取一定方法缓解故障，最终找出导致故障的根源问题。除了是在针对某个非常具体的组件进行安全审计的情况以外，我们不应该仅仅因为“某东西看起来有点问题”就发出警报。

紧急警报的处理会占用员工的宝贵时间。如果该员工正在工作时间段，该警报的处理会打断他原本的工作流程。如果该员工正在家，紧急警报的处理则会影响他的个人生活，甚至是把他从睡眠中叫醒。当紧急警报出现得太频繁时，员工会进入“狼来了”效应，怀疑警报的有效性甚至忽略该警报，有的时候在警告过多的时候甚至会忽略掉真实发生的故障。由于无效信息太多，分析和修复可能会变慢，故障时间也会相应延

长。高效的警报系统应该提供足够的信息，并且误报率非常低。

对监控系统设置合理预期

监控一个复杂的应用程序本身就是一项复杂的工程项目。即使在具有大量现成的基础设施的情况下，标记、收集、显示，以及报警这些工作，通常需要10～12个人组成的标准Google SRE团队中的1～2个人全职进行监控的构建和维护工作。由于我们花了很多精力将通用的监控基础设施进行了改造和集中化，这个数字已经随着时间下降了。但是目前每个SRE团队一般至少有一个“监控专员”（虽然平时看看流量图表可能很有意思，但是SRE团队通常会小心地避免任何需要某个人“盯着屏幕寻找问题”的情况。）

一般来说，Google趋向于使用简单和快速的监控系统配合高效的工具进行事后分析。我们会避免任何“魔法”系统—例如试图自动学习阈值或者自动检测故障原因的系统。这里可以举一个反例：检测最终用户请求速率的意外变化的系统。我们坚持监控系统规则越简单越好，同时要求这些监控规则可以检测某个非常简单、具体，但是严重的异常情况。监控数据的其他用处还包括容量规划、流量预测，用于这些方面的监控规则对错误和稳定性的要求更低，也就可以稍微复杂一些。针对某个试验功能的数据观测，可能时间跨度非常长（数月甚至数年），取样率也很低，这种用途可以容忍一定的错误率，因为这些偶尔出现的错误不会掩盖真正的长期趋势。

Google SRE在应对复杂依赖关系时的成功经验并不多。我们偶尔会使用这样的规则：“如果我知道数据库目前缓慢，那么发出一条数据库缓慢的警报，否则发出一条网站缓慢的警告”。针对依赖服务的监控规则，一般只用于系统中非常稳定的组件上。例如某个将用户流量从数据中心迁出的系统。“如果数据中心处于‘排水状态’（drained），那么不要发出有关延迟的警报”是一个常见的数据中心警报规则。由于Google基础设施的重构速度很快，很少有团队会在监控系统中维护复杂的依赖关系。

本章中讨论到的某些想法还有想象的空间：从现象到根源问题的定位速度可以更快，尤其是在一个不断改变的系统中。这一章给监控系统

设立了一些目标，同时提供了一些达到这些目标的方法。但是监控系统中最重要的一点就是整个“生产故障，人工处理紧急警报，简单定位和深入调试”过程必须要保持非常简单，必须能被团队中任何一个人所理解。

同样的，监控系统信噪比应该很高，发出紧急警报的组件需要非常简单而且可靠。产生警报的监控系统规则应该非常容易理解，同时代表一个清晰的故障场景。

现象与原因

监控系统应该解决两个问题：什么东西出故障了，以及为什么出故障。

“什么东西出故障了”即为现象（symptom）:“为什么”则代表了原因（可能只是中间原因，并不是根源问题）。表6-1列出了一些现象，以及它们对应的原因。

表6-1：现象与原因的示例

现象	原因
正在返回 HTTP 500 或者 404	数据库服务器拒绝连接
回复速度很慢	CPU 被某个排序操作占满了，某根网线被压在了机柜下面，造成断续的网络丢包
南极洲用户无法接收 GIF 动画	CDN 网络将某些 IP 列入了黑名单
私有内容可以被任何人访问	新的软件版本发布造成 ACL 丢失，允许所有请求进入

“现象”和“原因”的区分是构建信噪比高的监控系统时最重要的概念。

黑盒监控与白盒监控

Google大量依赖白盒监控，黑盒监控用得虽然不多，但都是在关键地方使用。黑盒监控与白盒监控最简单的区别是：黑盒监控是面向现象的，代表了目前正在发生的—而非预测会发生的—问题，即“系统现在有故障”。白盒监控则大量依赖对系统内部信息的检测，如系统日志、抓取提供指标信息的HTTP节点等。白盒监控系统因此可以检测到即将发生的问题及那些重试所掩盖的问题等。

这里应该注意，在一个多层系统中，某一个服务的现象是另外一个

服务的原因。例如，数据库性能问题。数据库读操作很缓慢是数据库SRE检测到的一个现象。然而，对前端SRE来说，他们看到的是网站缓慢，数据库读操作的缓慢则是原因。因此，白盒监控有时是面向现象的，有时是面向原因的，这取决于白盒系统所提供的信息。

当收集用于调试的遥测数据时，白盒监控是必不可少的。如果Web服务器看起来在那些依赖数据库的请求上很慢时，我们需要同时知道Web服务器检测到的数据库速度与数据库自己检测到的速度。否则，无法区分出到底是数据库问题还是Web服务器与数据库服务器之间的网络问题。

黑盒监控可以保证系统只在某个问题目前正在发生，并且造成了某个现象时才会发出紧急警报。另外一方面，针对那些还没有发生，但是即将发生的问题，黑盒监控通常是没用的。

4个黄金指标

监控系统的4个黄金指标分别是延迟、流量、错误和饱和度（saturation）。如果我们只能监控用户可见系统的4个指标，那么就应该监控这4个。

延迟

服务处理某个请求所需要的时间。这里区分成功请求和失败请求很重要。例如，某个由于数据库连接丢失或者其他后端问题造成的HTTP 500错误可能延迟很低。计算总体延迟时，如果将500回复的延迟也计算在内，可能会产生误导性的结果。但是，“慢”错误要比“快”错误更糟！因此，监控错误回复的延迟是很重要的。

流量

使用系统中的某个高层次的指标针对系统负载需求所进行的度量。对Web服务器来说，该指标通常是每秒HTTP请求数量，同时可能按请求类型分类（静态请求与动态请求）。针对音频流媒体系统来说，这个指标可能是网络I/O速率，或者并发会话数量。针对键值对存储系统来说，指标可能是每秒交易数量，或每秒的读取操作数量。

错误

请求失败的速率，要么是显式失败（例如HTTP 500），要么是隐式失败（例如HTTP 200 回复中包含了错误内容），或者是策略原因导致的失败（例如，如果要求回复在1s内发出，任何超过1s的请求就都是失败请求）。当协议内部的错误代码无法表达全部的失败情况时，可以利用其他信息，如内部协议，来跟踪一部分特定故障情况。监控方式也非常不一样：在负载均衡器上检测HTTP 500请求可能足够抓住所有的完全失败的请求，但是只有端到端的系统才能检测到返回错误内容这种故障类型。

饱和度

服务容量有多“满”。通常是系统中目前最为受限的某种资源的某个具体指标的度量。（在内存受限的系统中，即为内存；在I/O受限的系统中，即为I/O）。这里要注意，很多系统在达到100% 利用率之前性能会严重下降，增加一个利用率目标也是很重要的。

在复杂系统中，饱和度可以配合其他高层次的负载度量来使用：该服务是否可以正常处理两倍的流量，是否可以应对10%的额外流量，或者甚至应对当前更少的流量？对没有请求复杂度变化的简单服务来说（例如，“返回一个随机数”服务，或者是“返回一个全球唯一的单向递增整数”服务），根据负载测试中得到的一个固定数值可能就足够了。但是正如前文所述，大部分服务都需要使用某种间接指标，例如CPU利用率，或者网络带宽等来代替，因为这些指标通常有一个固定的已知的上限。延迟增加是饱和度的前导现象。99% 的请求延迟（在某一个小的时间范围内，例如一分钟）可以作为一个饱和度早期预警的指标。

最后，饱和度同样也需要进行预测，例如“看起来数据库会在4个小时内填满硬盘”。

如果我们度量所有这4个黄金指标，同时在某个指标出现故障时发出警报（或者对于饱和度来说，快要发生故障时），能做到这些，服务的监控就基本差不多了。

关于长尾问题

构建监控系统时，很多人都倾向于采用某种量化指标的平均值：延迟平均值，节点的平均CPU使用率，数据库容量的平均值等。后两个例子中存在的问题是很明显的：CPU和数据库的利用率可能波动很大，但是同样的道理也适用于延迟。如果某个Web服务每秒处理1000个请求，平均请求延迟为100ms。那么1%的请求可能会占用5s时间。[10]如果用户依赖几个这样的服务来渲染页面，那么某个后端请求的延迟的99%可能就会成为前端延迟的中位数。

区分平均值的"慢"和长尾值的"慢"的一个最简单办法是将请求按延迟分组计数（可以用来制作直方图）：延迟为0～10ms之间的请求数量有多少，30～100ms之间，100～300ms之间等。将直方图的边界定义为指数型增长（这个例子中倍数约为3）是直观展现请求分布的最好方式。

度量指标时采用合适的精度

系统的不同部分应该以不同的精度进行度量，例如：

- 观察1分钟内的CPU平均负载可能会错失导致长尾延迟过高的某种较长时间的CPU峰值现象。
- 对于一个每年停机时间小于9小时的Web服务来说（年度可用率99.9%），每分钟检测1次或2次的监控频率可能过于频繁。
- 对目标可用率为99.9% 的某个服务每 1 分钟或者2分钟检查一次硬盘剩余空间可能也是没必要的。

应该仔细设计度量指标的精确度。每秒收集CPU负载信息可能会产生一些有意思的数据，但是这种高频率收集、存储、分析可能成本很高。如果我们的监控目标需要高精度数据，但是却不需要极低的延迟，可以通过一些内部采样机制外部汇总的方式降低成本。

例如：

1.将当前CPU利用率按秒记录。

2.按 5%粒度分组，将对应的CPU利用率计数+1。

3.将这些值每分钟汇总一次。

这种方式使我们可以观测到短暂的CPU热点，但是又不需要为此付出高额成本进行收集和保留高精度数据。

简化，直到不能再简化

将之前讨论的所有需求累加起来可能会形成一个非常复杂的监控系统—以下是几个复杂度的例子：

- 不同的延迟阈值，在不同的百分位上，基于各种各样不同的指标进行报警。
- 检测和揭示可能的故障原因。
- 给每种可能的原因构建对应的监控台页面。

复杂是没有止境的。就像任何其他软件系统一样，监控系统可能会变得过于复杂，以至于经常出现问题，变更非常困难，维护起来难度很大。

因此，设计监控系统时一定要追求简化。在选择需要检测什么的时候，将下列信息记在心里：

- 那些最能反映真实故障的规则应该越简单越好，可预测性强，非常可靠。
- 那些不常用的数据收集、汇总，以及警报配置应该定时删除（某些SRE团队的标准是一个季度没有用到一次即将其删除）。
- 收集到的信息，但是没有暴露给任何监控台，或者被任何警报规则使用的应该定时删除。

在Google的经验里，指标的收集和汇总，加上警报系统与监控台系

统，作为一个相对独立的系统运行是比较好的。（事实上，Google的监控系统是作为多个二进制文件运行的，但是通常人们把它们当成一个整体来学习。）将监控系统与其他的复杂系统操作结合起来（例如系统性能统计，单独进程的调试，异常或者崩溃的跟踪，负载测试，日志收集和分析，流量检测等）会导致监控系统过于复杂，容易出现问题。和其他软件工程的理念一样，保持系统相对独立，清晰简单，松耦合的接口是更好的策略（例如，利用Web API来收集性能数据，采用一种可以持续很久不变的简单数据格式）。

将上述理念整合起来

本章描述的理念整合起来就成为Google SRE广泛接受和遵循的监控与警报设计哲学。虽然这个设计哲学有一定理想性，但是书写和评审某个新警报时可以依赖的好方法。该哲学同时有助于鼓励团队在解决问题时向正确的方向进行。

当为监控系统和警报系统增加新规则时，回答下列问题可以帮助减少误报：[11]

- 该规则是否能够检测到一个目前检测不到的、紧急的、有操作性的，并且即将发生或者已经发生的用户可见故障？[12]

- 是否可以忽略这条警报？什么情况可能会导致用户忽略这条警报，如何避免？

- 这条警报是否确实显示了用户正在受到影响？是否存在用户没有受到影响也可以触发这条规则的情况？例如测试环境和系统维护状态下发出的警报是否应该被过滤掉。

- 收到警报后，是否要进行某个操作？是否需要立即执行该操作，还是可以等到第二天早上再进行？该操作是否可以被安全地自动化？该操作的效果是长期的，还是短期的？

- 是否也会有其他人收到相关的紧急警报，这些紧急警报是否是不必要的？

以上这些问题其实反映了在应对紧急警报上的一些深层次的理念：

- 每当收到紧急警报时，应该立即需要我进行某种操作。每天只能进入紧急状态几次，太多就会导致“狼来了”效应。

- 每个紧急警报都应该是可以具体操作的。

- 每个紧急警报的回复都应该需要某种智力分析过程。如果某个紧急警报只是需要一个固定的机械动作，那么它就不应该成为紧急警报。

- 每个紧急警报都应该是关于某个新问题的，不应该彼此重叠。

从这种角度出发，我们可以得出以下结论：如果某个紧急警报满足上述四点，那么不论是从白盒监控系统还是黑盒监控系统发出都一样。最好多花一些时间监控现象，而不是原因。针对“原因”来说，应该只监控那些非常确定的和非常明确的原因。

监控系统的长期维护

在现代生产环境中，监控系统需要跟随不断演变的软件一起变化，软件经常重构，负载特性和性能目标也经常变化。现在的某个不常见的、自动化比较困难的警报可能很快就会变成一个经常触发、需要一个临时的脚本来应对的问题。这时，某个人应该去寻找和消除背后的根源问题：如果这种解决办法不可行，那么这条警报的应对就必须要完全自动化。

关于监控系统的设计决策应该充分考虑到长期目标。今天发出的每个紧急警报都会占用优化系统的时间，所以经常会牺牲一些短期内的可用性和性能问题，以换取未来系统性能的整体提升。以下是两个具体的案例。

Bigtable SRE：警报过多的案例

Google　内部的基础设施通常提供某个SLO（参见第4章），并且伴随有相应的SLO监控。很多年以前，Bigtable　SLO是基于某个假想的客户端的平均性能得出的。由于Bigtable和底层存储技术栈中的一些问

题，平均性能受很大的“长尾”所影响：请求中最差的5%　比其他的请求要慢很多倍。

当接近SLO目标的时候，系统会发出E-mail警报；当低于SLO目标时，系统则会发出紧急警报。这两种警报一旦触发都是数量巨大的，将消耗工程师非常多的时间来处理：团队花费了很多时间来甄别每条警报，以便找出那些真正可以应对的警报。同时，我们经常会错过那些真正影响用户的警报，因为大部分的警报都不会影响用户。很多紧急警报其实并不紧急，因为基础设施中存在一些大家都知道的问题，所以这些警报经常没人回复，或是回复得很消极。

为了解决这个问题，团队采用了三阶段方式：首先，在重点提升Bigtable性能的同时，团队临时将SLO目标降低，采用了请求延迟的75%百分位作为SLI。同时关闭了E-mail警报，因为数量巨大，一个一个检测是不可能的。

该策略给团队带来了喘息空间，提供了一些时间修复Bigtable的长期问题，以及底层的存储技术栈，而不是不停地修复一些战术性问题。on-call工程师可以真正做一些事情，而不是整天被紧急警报打断。最终，通过对警报策略的临时性调整使得团队可以更好地优化服务，在未来提供更好的服务质量。

Gmail：可预知的、可脚本化的人工干预

在Gmail服务的早期，整个服务是基于一个分布式进程管理系统构建的，称为“Workqueue”，该系统最初是为搜索索引批处理任务构建的。Workqueue针对长期运行的进程做了一些适应，最终应用于Gmail。但是相对不透明地调度代码中的某些Bug一直没有被消除。

当时，Gmail的监控系统会在某个任务被Workqueue系统干掉时发出警报。即使是在当时，这种配置也很差劲：Gmail有几千个任务，每个任务都代表了百分之几的用户。虽然我们非常关注Gmail用户的体验，但是这样的警报规则是不可持续的。

为了解决这个问题，Gmail　SRE构建了一个工具来操作调度器尽量

减小对用户的影响。整个团队就这个问题进行了多次讨论：是否应该自动化整个方案？但是一些团队成员坚持认为开发这样的自动化方案会影响对真实问题的最终修复。

团队中出现的这种冲突是合理的，这反映出团队自我约束方面的信任危机：一些团队成员想要实现某种“hack”从而为真正的修复方案争取时间，而另外一些成员则担心实现这个“hack”会使得真正的修复优先级无限降低。这种担心是正确的，确实这种打补丁的形式会造成无法维护的系统债务。管理者和技术领导者在这个过程中应该起到直接作用，在紧急警报带来的压力减轻之后应该继续支持和优先处理那些长期修复问题的工作。

任何一个可以死记硬背，或者基于某种公式的紧急警报的响应都应该引起注意。团队中一部分人不愿意打补丁的原因是因为他们不相信未来能够处理这些技术债务。这是一个值得与上级讨论的问题。

长跑

上面的Gmail和Bigtable的例子有个共同点：短期与长期的可用性的冲突。经常，通过一些“暴力”因素，可以使一个摇摇晃晃的系统保持一定的高可用性。但是这种方案通常是不能持久的，而且这经常依赖于某个团队成员的个人英雄主义。短期内，接受某种可控的可用性的降低可以换取一些系统长期性的提升。将所有紧急警报作为一个整体来审视是很重要的，要考虑当前的紧急警报的级别是否对未来的一个高可用、高可靠性的系统有帮助。Google管理团队会按季度进行紧急警报的频率统计（经常以每次on-call轮值发生的故障数量统计，某个故障可能由多个紧急警报组成），保证每个决策者都理解目前的运维压力，以及系统的健康状况。

小结

健康的监控和警报系统应该是非常简单、易于理解的。紧急警报应该关注于现象，针对原因的一些启发性分析应该作为调试过程的补充，而不应该进行报警。监控的技术栈层面越高，监控现象越容易，但是监控某些子系统（如数据库）的饱和度和性能参数可能要在该子系统内部

直接进行。E-mail警报的价值通常极为有限，很容易变成噪声。我们应该倾向于构建一个良好的监控台页面，直接显示所有的非紧急的异常情况。

长远来看，要建立一个成功的on-call轮值体系，以及构建一个稳定的产品需要选择那些正在发生和即将发生的问题来进行报警，设置一个可以实际达到的合理目标，保证监控系统可以支持快速的问题定位与检测。

第7章 Google 的自动化系统的演进

作者：**Niall Murphy**、**John Looney**、**Michael Kacirek**

编辑：**Betsy Beyer**

“黑科技”之外，就只剩自动化和机械化了。

——Federico García Lorca（1898–1936），西班牙诗人和剧作家

对于SRE而言，自动化是一种力量倍增器，但不是万能药。当然，对力量的倍增并不能改变力量用在哪的准确性：草率地进行自动化可能在解决问题的同时产生出其他问题。因此，虽然我们认为在大多数情况下以软件为基础的自动化是优于手动操作的，但是比这两个选择更好的方案是一个不需要这些的系统设计— 一个自治的系统。或者换一种方式来看，自动化的价值不仅来源于它所做的事情，还包括对其的明智应用。我们将讨论自动化系统的价值，以及SRE对自动化系统的态度的演进历史。

自动化的价值

自动化的价值究竟是什么？[13]

一致性

在应对系统规模的增加之外，还有许多其他的理由来使用自动化。

以大学计算系统为例，许多系统工程师都是从这里开始他们的职业生涯的。具有该背景的系统管理员一般负责运维一系列的物理机或软件，并且非常习惯于在履行职责的过程中手动执行各种操作。一个常见的例子是创建用户账户；其他例子包括单纯的操作职责，如确保备份正确进行、进行故障迁移和一些小的数据修改，例如修改上游DNS服务器的resolv.conf，修改DNS服务器的区域数据，以及类似的操作。然而，最终看来，这种手动执行任务的方式对于整个组织和实际执行的人都不好。首先，任何一个人或者一群人执行数百次动作时，不可能保证每次都用同样的方式进行：没有几个人能像机器一样永远保持一致。这种不可避免的不一致性会导致错误、疏漏、数据质量的问题和可靠性问题。在这个范畴内——一致性地执行范围明确、步骤已知的程序——是自动化的首要价值。

平台性

自动化不仅仅提供一致性。通过正确地设计和实现，自动化的系统可以提供一个可以扩展的、广泛适用的，甚至可能带来额外收益的平台。[14]（相对来看，不进行自动化既不符合成本收益，也无法扩展，就像是在系统运维过程中额外交付的税务。）

一个平台同时也将错误集中化了。也就是说，在代码中修复某个错误可以保证该错误被永远修复。一个平台更容易地被扩展，从而执行额外的任务，这比教授人类要容易得多。（有的时候教育人类意识到必须进行某种操作很难）。取决于任务的性质，平台可能能比人类更持续或者更频繁地运行任务，甚至完成一些对于人类而言并不方便执行的任务。此外，一个平台可以暴露自身的性能指标，也可以帮助你发现流程中以前所不知道的细节，这些细节在平台范围内更容易衡量。

修复速度更快

采用自动化系统解决系统中的常见故障，可以带来额外的好处（这常常是SRE构建自动化系统的原因）。如果自动化能够始终成功运行，那么就可以降低一些常见故障的平均修复时间（MTTR）。随后，用户可以把时间花在其他任务上，从而提高开发速度。因为用户不再需要花费时间来预防问题发生或者进行（更常见的）事后清理。

在行业内普遍认同的是，在产品生命周期中一个问题越晚被发现，修复代价越高；参见第17章。一般来说，解决实际生产中出现的问题是最昂贵的，无论是时间还是金钱方面，这意味着，构建一个在问题发生之后马上应对的自动化系统，对于降低系统的总成本非常有利，前提是该系统足够大。

行动速度更快

在基础设施中，SRE 自动化系统应用广泛。这是因为人类通常不能像机器一样快速反应。例如，故障转移或流量调整对于一个特定的应用程序来说可以被很好地定义，要求一个人间歇性地手动按一个叫“允许系统继续运行”的按钮是没有任何意义的。（确实，有时自动化程序可以最终使一个坏的情况变得更糟，但是这恰恰是将这种程序的范围明确定义的原因。）Google拥有大量的自动化系统；在很多情况下，没有自动化参与我们所支持的服务，这些服务就不能长久运行，因为它们很久之前就超越了人工操作所能管理的门槛。

节省时间

最后，节省时间是一个经常被引用的使用自动化的理由。虽然大家经常选择这个依据来支持自动化，但是在很多情况下这种优势不能立即计算出来。工程师对于一个特定的自动化或代码是否值得编写而摇摆不定，不停地比较写该代码所需要花费的精力与不需要手动完成任务所节省的精力。[15]这里很容易忽略的一个事实是，一旦你用自动化封装了某个任务，任何人都可以执行它们。因此，时间的节省适用于该自动化适用的所有人。将某个操作与具体操作的人解耦合是很有效的。



Joseph Bironas，负责Google数据中心集群上线流程的SRE，有力地提出：

“如果我们持续产生不可自动化的流程和解决方案，我们就继续需要人来进行系统维护。如果我们要雇佣人来做这项工作，就像是在用人类的鲜血、汗水和眼泪养活机器。这就像是一个没有特效但是充满了愤怒的系统管理员的Matrix世界。”

自动化对Google SRE的价值

所有上文提到的这种益处和弊端对SRE来说也同样适用，Google更倾向于自动化。我们对于自动化偏爱部分来自于Google特有的业务挑战：Google的产品和服务是全球部署的，而我们通常也没有时间和其他组织一样手动运维系统。[16]对于真正的大型服务来说，一致性、快速性和可靠性这些因素主导了大多数有关自动化的权衡的讨论。

支持自动化的另一种说法，特别是在Google内部，是我们在第2章描述的，Google内部复杂却令人惊讶的高度统一的生产环境。虽然其他组织可能运行一个没有现成的API的重要设备，或者一个没有源代码的软件，以及没有对生产运维完全的控制权的情况，

Google通常可以避免这些情况发生。Google为那些供应商没有提供API的系统构建了自己的API。虽然购买某一特定任务的软件会更便宜，但我们仍倾向于选择自主开发，因为这样可以产生具有长期价值的API。我们花了很多时间来克服自动化系统管理的各种障碍，随后坚决地进行了自动化系统管理本身的开发。鉴于Google管理其源代码的方式（参见文献[Pot16]），几乎所有SRE接触的系统的源代码都是多多少少可用的。这也意味着我们的使命“在生产环境中拥有产品”要更加容易，因为我们控制了全部的技术栈。

当然，尽管Google在思想上倾向于尽可能使用机器管理机器，但实际情况需要一定的变通。将每个系统的每一个组件都自动化是不合适的，同时不是所有人都有能力或者倾向于在一个特定的时间开发自动化系统。我们的一些基本系统由快速原型开始，并不是为了长久运行，也不是为自动化而设计的。前面一段阐述了对我们的观点的最大化情况，也是我们在Google的范围内付诸行动取得了广泛的成功的一种观点。一般来说，我们在可能的情况下都选择了创建平台，或者是将我们自己“定

位”在未来可以创建平台的位置上。我们认为这个以平台为基础的方法对于可管理性和可扩展性是非常有必要的。

自动化的应用案例

在运维行业中，自动化这个术语一般用来指代通过编写代码来解决各种各样的问题。尽管写这些代码的动机以及最终产生的解决方案本身往往区别很大。更广泛地说，在这一观点中，自动化是“元软件”，也就是操作其他软件的软件。

正如我们之前暗示的，自动化有许多用例。下面是一个非详尽的例子列表：

- 创建用户账户。
- 某个服务在某个集群中的上线和下线过程。
- 软件或硬件安装的准备和退役过程。
- 新软件版本的发布。
- 运行时配置的更改。
- 一种特殊情况的运行时配置更改：依赖关系的更改。

这个列表基本上可以无限扩展。

Google SRE的自动化使用案例

在Google内部，上述所有使用案例都有，甚至更多。

然而，在Google SRE内部，我们主要负责运维基础设施，而不是管理那些穿过基础设施的数据的质量。这条分界线并不是完全清晰的——例如，我们会非常关注某个数据集在推送之后消失一半的情况，因此我们会针对这种粗粒度的差异报警。但对于SRE而言，具体修改系统中一些账户的某个子集的属性是相当罕见的。因此，自动化的情境通常是自动化管理系统的生命周期，而非系统内部的数据：例如，部署新的群集

服务。

即使如此，SRE的自动化努力也与其他组织所做的差距并不大。SRE使用不同的工具来管理自动化，同时关注的重点不同（我们接下来将会对此进行讨论）。

广泛使用的工具有Puppet、Chef、cfengine，甚至 Perl都提供了自动化完成特定任务的方法，主要区别在于对帮助进行自动化的组件的抽象层次不同。Perl这种完整的语言提供了POSIX级别的接口，理论上在系统API层面上提供了一个基本上是无限的扩展范围。[17]而Chef和Puppet则提供了一些“开箱即用”的抽象层，通过对这些抽象层的操作可以直接操作服务或者其他高级对象。这里的妥协十分经典：高层次的抽象更容易管理和进行逻辑推理，但当你遇到一个“有漏洞的抽象”时，就会系统地、重复地甚至不一致地出现故障。例如，我们经常假设，将一个新的二进制文件发布到集群中是原子性的；该集群最终会全部变成新版本，或者全部维持旧版本。然而，现实中的行为很复杂：集群网络中途可能会发生故障；物理机可能发生故障；与集群管理层的通信可能会失败，使系统进入不一致的状态；视具体情况不同，新的二进制文件可能安装了，但没有推送，或者推送了但是没有启动，或者启动了但是无法验证。没有几个抽象模型能够成功地模拟这些结果，大部分模型都会中止并要求人工干预。而那些真正糟糕的自动化系统甚至都不会这样做。

SRE在自动化领域有一系列设计哲学和产品，它们中的一些类似于一种不会特别详细地对高层次实体建模的通用部署工具，另外一些则类似于在非常抽象的层次上描述服务部署的语言。后者往往比前者更加通用，更符合通用平台。然而，我们生产环境的复杂性有时意味着前者是更容易采用的选择。

自动化分类的层次结构

虽然所有这些自动化步骤都是有价值的，同时自动化平台本身也是很有价值的。在一个理想的世界里，我们不需要任何平台之外的自动化进程。事实上，构建一个完全不需要胶合逻辑的系统要比有一个依赖外部的胶合逻辑的系统更好，不仅仅是因为内部化效率更高（尽管这样的

效率提升很有价值），而是因为设计中将胶合逻辑排除在外了。这样做需要将胶合逻辑的具体用例—一般来说是对系统的直接操作，例如添加账户或执行系统集群上线—用某种方法在应用程序中直接处理。

这里提供了一个更详细的例子，Google的集群上线系统自动化经常出现问题，因为这些最终是在核心系统之外维护的，因此经常受到“代码腐烂”（bit rot）的影响。也就是说，当基本系统变化的时候，上线自动化系统没有随之改变。尝试将两者（集群上线自动化系统和核心系统）更紧密地连接起来的努力常常会由于两个团队不一致的优先级定义而失败。产品的研发人员会—不能说不合理的—抵制对每一次改动都进行一次测试部署的要求。其次，关键性的，但是不经常执行的，以至于很难测试的自动化系统尤其脆弱，因为反馈的周期很长。集群故障转移自动化是一个经典例子：故障转移可能每隔几个月甚至更长时间才发生一次，导致每次执行都不一致。自动化的演进遵循以下路径：

1）没有自动化

手动将数据库主进程在多个位置之间转移。

2）外部维护的系统特定的自动化系统

SRE在他或她的主目录中保存了一份故障转移脚本。

3）外部维护的通用的自动化系统

SRE将数据库支持添加到了每个人都在使用的“通用故障转移”脚本中。

4）内部维护的系统特定的自动化

数据库自己发布故障转移脚本。

5）不需要任何自动化的系统

数据库注意到问题发生，在无须人工干预的情况下进行故障转移。

SRE讨厌手动操作，所以我们尽力创造不需要他们的系统。然而，

有时手动操作是不可避免的。

同时，相对于在特定系统相关配置上变更的自动化，另外一种自动化是面向整个生产领域的变更。在Google这种高度集中的专有生产环境下，有大量的跨特定服务范围的变更存在—比如，对上游的Chubby服务器的变更，使访问更可靠的一个Bigtable客户端库的功能开关的变更等—这些变更也需要安全地管理，在必要的情况下进行回退。当变更数量超过一定量之后，这种全生产范围内的变更就不可能手动完成了，即使在这之前，对一个变更很小或者通过基本的重启+检查就可以完成的流程进行人工监督是毫无意义的。

下面让我们使用内部案例研究来详细描述上文提到的这些要点。第一个案例研究的是如何利用有远见的工作来成功地实现了SRE自我标榜的涅槃：通过自动化将SRE从整个流程中消除。

让自己脱离工作：自动化所有的东西

很长一段时间以来，Google的广告产品将数据存储于一个MySQL数据库中。因为广告数据显然需要很高的可靠性，一个SRE团队负责管理那些基础设施。从2005年到2008年，我们认为广告数据库基本是在一种成熟的和可管理的状态下运行的。例如，我们已经将标准副本替换流程的常规工作中最糟糕的部分自动化掉了，但是没有将全部工作自动化。我们认为广告数据库已经管理得很好了，已经摘到了优化和规模方面的大部分的“低挂的果实”。然而，随着日常工作变得越来越容易，SRE团队成员开始考虑下一级的系统研发：将MySQL迁移到Google的集群调度系统Borg之下。

我们希望这种迁移会带来两点主要益处：

- 彻底消除对物理机/数据库副本的维护：Borg会自动安装新任务，或者重启出问题的任务。

- 将多个MySQL实例安装在同一台物理机上：利用容器化可以更好地利用计算机资源。

2008年年底，我们成功地在Borg上部署了一个原型实例。然而不幸

的是，这带来了一个严重的新增困难。Borg的一个核心操作特点就是它的任务可以自动地移动：Borg内部运行的任务，通常的移动频率达到了每周一次或两次。这样的频率对数据库的副本来说是可以接受的，对于主实例则不可接受。

当时，主实例故障转移过程每次需要30～90分钟。由于我们在共享的机器上运行，同时需要进行内核升级而重启，我们每周都需要在正常的物理机故障率之外进行一些无关的故障转移。这一因素，加上系统中分片数量庞大的因素，意味着：

- 手动故障转移将消耗大量的人力时间，并且在最好条件下也只能给我们99%的可用性，这样比产品的实际业务需要低很多。

- 为了满足错误预算，每个故障转移的停机时间要小于30s。优化依赖人为操作的流程达到这一目标是不可能的。

因此，我们唯一的选择是自动进行故障转移。实际上，我们不仅仅需要自动化故障转移流程。

2009年，广告SRE完成了自动故障切换后台程序，称为"决策者"。决策者程序可以在95%的时间内用小于30s的时间完成计划内和计划外的MySQL数据库故障转移流程。随着决策者的诞生，MySQL On Borg（MOB）最终变成了现实。我们从不断优化基础设施以避免进行故障转移的模式转化为拥抱失败，承认故障是不可避免的，并因此通过自动化进行快速恢复的模式。

虽然通过自动化，我们可以在一个每周强制两次重启的世界里仍然保证MySQL的高可用，但是这也带来了它自己的一套成本，所有的应用程序都必须增加很多错误处理逻辑。由于MySQL开发世界中的常态是假设MySQL实例是整个技术栈中最稳定的成分，这种改变意味着我们要定制类似JDBC这样的软件，以便对我们的故障环境更加宽容。然而，与决策者一起迁移到MoB上的益处对这些成本来说是很值得的。迁移到MoB之后，我们的团队在无聊的运维任务上花费的时间下降了95%。整个故障转移过程是自动化的，所以单个数据库任务中断不再给任何人发出紧急报警。

这一新的自动化的主要好处是，我们有更多的空闲时间花在改进基础设施的其他部分上。这些改进有一个连锁效应：节省的时间越多，优化和自动化其他烦琐工作的时间就越多。最终，我们能够将数据库结构的变更也自动化了，这导致广告数据库的总运维成本下降了近95%。有人可能会说，我们已经成功地自动将自己从这项工作中自动化出来了。同时，在硬件资源方面也有了改进。迁移到MoB的过程中释放了大量的资源，因为我们可以在同样的机器上运行多个MySQL实例，从而提高了硬件利用率。总的看来，我们最终解放了近60%的硬件资源，团队的硬件资源和工程资源都非常充足。

该例子描述了通过额外努力构建一个平台，而不是仅仅取代现有的手动流程的好处。下一个例子来自于集群基础设施组，描述了一些进行全自动化时可能遇到的更困难的权衡。

舒缓疼痛：将自动化应用到集群上线中

十年前，集群基础设施SRE团队似乎每隔几个月都要雇用新人。事实上，大约与我们新集群上线的频率相同。因为在新集群中启动新的服务能够让新员工接触到服务内部的信息，这个任务似乎是一个自然的和有效的培训工具。

为使一个集群达到可用状态所需采取的步骤如下：

1.给数据中心大楼装配电源和冷却设备。

2.安装和配置核心交换机和与主干网的连接。

3.安装几个初始的服务器机架。

4.配置一些基本服务，例如DNS和安装器（installer），然后配置Chubby锁服务、存储服务和计算服务。

5.部署剩余的机架。

6.给面向用户的服务分配资源，使其团队可以建立服务。

第4步和第6步是极为复杂的。虽然基本的服务，如DNS是相对简单

的，但存储和计算子系统当时正在全力开发中，所以每周都会增加新的功能开关、组件和优化。

一些服务有超过一百个不同的组件子系统，每一个都具有复杂的网状依赖。某个子系统的配置失败，或者是没有按照其他集群来配置一个系统或组件，将造成潜在的故障发生。

在一个案例中，一个存有数个PB的Bigtable集群因为延迟原因被配置为不使用12个磁盘系统中的第1个（日志型）磁盘。一年后，一些自动化程序假设如果一台机器的第一个磁盘没有被使用，这台机器就不被配置任何存储服务；因此，可以安全地清除数据。这导致所有的Bigtable数据立刻被清除了。幸好我们的数据有多个实时备份，但是这种意外也是很糟糕的。自动化应该对那些隐含的“安全”信号非常小心。

早期的自动化关注于加速集群交付。这种方法往往依靠“有创意”地使用SSH来应对繁琐的包分发和服务初始化问题。采用这种战略一开始很成功，但是这些格式自由的脚本逐渐堆积形成了技术债务。

使用Prodtest检测不一致情况

随着集群的数量增长，一些集群需要手工调整功能开关和配置。这样做的结果是，团队在追逐难以发现的错误上浪费了越来越多的时间。如果某个使GFS对日志处理响应更快的开关泄露到了默认模板中，文件很多的集群就可能在负载之下内存超标。每次大型配置改变时，令人恼怒和极为耗时的错误都会偷偷潜入。

我们用来配置集群的、有创造性的—但是脆弱的—shell脚本既无法适应需要修改该脚本的人的数量增加，也不能适应需要构建的集群的大量组合形式。这些shell脚本同时在宣布服务状态良好，可以承受用户的流量的问题上也未能解决一些显著问题，比如：

- 是否所有服务的依赖关系都可用，并且正确地配置了？
- 所有的配置和包都与其他部署一致吗？
- 团队是否能够确认配置中的每个例外都是合理的？

Prodtest（生产测试）是对这些问题的一个巧妙的解决方案。我们对Python单元测试框架进行了扩展，使其可以用来对实际服务进行单元测试。这些单元测试有依赖关系，允许进行链条式测试，一个测试中出现的故障可以很快中止整个测试。以图7-1所示的测试为例。

某个团队的Prodtest 执行时，会传入该集群的名字，这样它可以验证该集群中的服务。后来增加功能可以让我们生成一个有关单元测试和它们状态的图表。这个功能使工程师能够更快看到他们的服务是否在所有集群中都被正确配置了，如果没有，为什么没有。图中突出了出错的步骤，以及出现故障的单元测试会输出的更详细的错误信息。

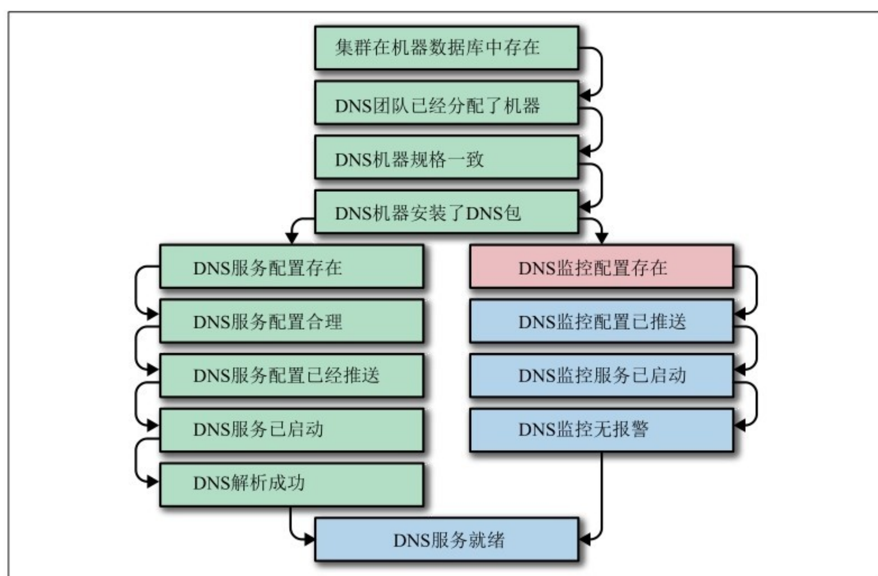


图7-1:DNS服务的 ProdTest，展示了一个测试出现问题后如何中止之后的一系列测试。

每当一个团队遇到因另一个团队意外错误配置而导致的延迟时，可以提交一个Bug来扩展Prodtest。这确保了类似问题以后可以尽早发现。SRE能够非常自豪地向用户保证，他们的所有服务—不管是配置的新服务还是现有服务—都可以可靠地服务生产业务。

我们的项目经理第一次可以准确地预测一个集群的“上线的时间”了，同时可以对为什么每个集群从“网络就绪”到“服务线上流量”需要6个星期甚至更长时间有了一个完整的理解。毫无准备的，SRE突然从管理高层收到任务：在3个月内，有5个新的集群将在同一天进入网络就绪

的状态。请在一周内完成上线。

幂等地解决不一致情况

“一周内上线”是一个很恐怖的任务。我们目前有十几个团队编写出的几万行的shell脚本。虽然我们能够快速分析任何一个集群的问题所在，但是修复它意味着十几个团队不得不提交数百个Bug，接着我们就只能希望这些Bug能被及时地解决。

我们意识到，从“Python单元测试发现错误配置”到“Python代码修复错误配置”的演进能够使我们更快解决这些问题。

单元测试已经能够指出被检查的集群中哪个测试失败，我们就给每个测试增加了一个修复程序。如果每一个修复程序都具有幂等性，并且假设所有的依赖关系都得到满足，那么修复问题就是简单而安全的。强调修复程序的幂等性意味着团队可以每15分钟运行一次他们自己的“修复脚本”，而不用担心对集群配置造成损害。如果DNS团队的测试在等待机器数据库团队对新集群的配置，只要该集群出现在数据库中，DNS团队的测试和修补程序就会立刻开始工作。

以图7-2　展示的测试为例。如果TestDNSMonitoringConfigExists失败，我们可以调用FixDNSMonitoringCreateConfig，这样会从一个数据库中将配置去掉，然后将一个骨架配置文件提交到版本控制系统中。接下来重试TestDNSMonitoringConfigExists成功，然后就可以进行TestDNSMonitoringConfigPushed测试。如果测试失败，就运行FixDNSMonitoringPushCon☆g。如果一个修复程序失败多次，自动化系统假设这个修补程序失败，并且会停止进行，通知用户修复。

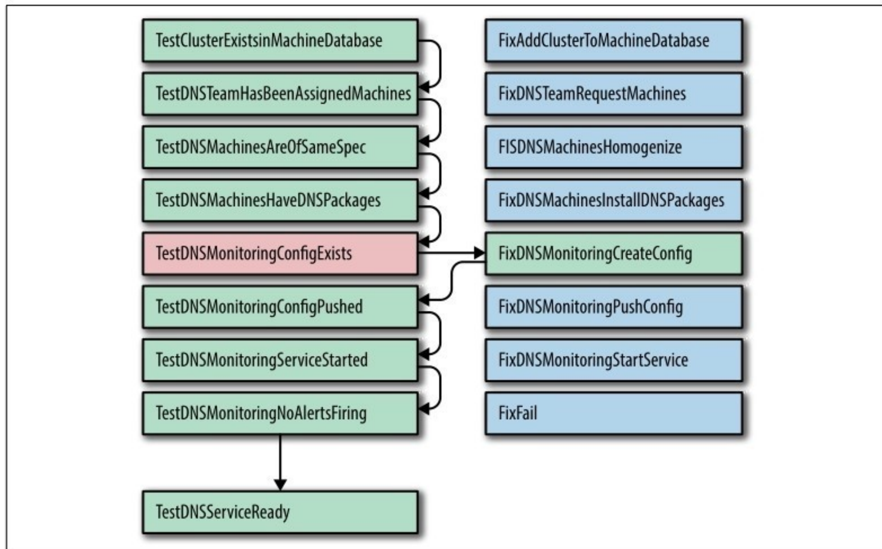


图7-2:DNS 服务的ProdTest，体现了一个失败测试的后果仅仅是多运行一个修复程序。左侧是测试，右侧是对应的修复程序。

有了这些脚本后，一小组工程师可以保证我们能够在一个或两个星期内从“网络就绪以及机器在数据库中存在”发展到“服务网络搜索和广告流量的1%流量”。在当时，这似乎达到了自动化技术的巅峰。

然而，现在回头来看，这种方式本身是存在严重缺陷的；在测试、修复、再测试之间的延迟引入了“不稳定”的测试，时好时坏。并不是所有的修复程序都具有幂等性，所以一个“不稳定”的测试引起的一次修复可能造成系统处于不一致的状态。

专业化倾向

自动化程序的不同体现在三个方面：

- 能力，即准确性。
- 延迟，开始执行后，执行所有步骤需要多久。
- 相关性，自动化所涵盖的实际流程比例。

最开始，我们的流程有如下特点：能力强（由服务的所有者维护和运行）、高延迟（服务拥有者在他们的业余时间执行流程或者分配给新

工程师来做），并且相关性高（服务拥有者知道现实世界已经改变并且能够修复自动化）。

为了减少集群上线的延迟，许多服务团队指导一个单独的“集群上线”团队来运行自动化。这个团队使用工单来启动每个阶段，这样我们就能够跟踪余下的任务，以及这些任务分配给了谁。如果关于自动化模块之间的人际互动发生在同一个房间里的人与人之间，集群上线就可以在更短的时间内发生。最后，我们的流程变成了能力强的、精确的，以及及时的自动化流程！

但是这个状态没有持续太久。现实世界是混乱的：软件、配置、数据等都发生了变化，这导致受影响的系统每天产生超过1000个独立变化。最受自动化Bug所影响的人不再是领域专家，因此自动化的相关性下降了（这意味着新的步骤被遗漏了），能力减弱了（新的功能开关可能导致自动化失败）。然而，这种质量的下降一段时间之后才会影响速度。

自动化代码和单元测试代码一样，当维护团队不再关心它与它所覆盖的代码仓库同步的时候就会逐渐死去。整个世界都在围绕代码改变：DNS的团队增加了新的配置选项，存储团队改变了他们的包名，网络团队需要支持新的设备。

通过消除运维相关服务的团队维护和运行自动化代码的责任，我们创造了一个不合理的组织激励机制：

- 某个最主要任务是加速现存的集群上线的团队是没有动力去减少服务运维团队在生产流程后期运维服务产生的技术负债的。

- 一个不亲自运行自动化的团队是没有动力去建设一个能够很容易自动化的系统的。

- 一个产品经理的时间表如果不受低质量的自动化影响，他将永远优先新功能的开发，而不是简化和自动化。

最可用的工具通常是由那些每天使用它们的人写成的。类似的论点是，产品研发团队将会在保留一些生产运维知识的过程中受益。

集群上线流程再一次成为了高延迟的、不准确的、低能力的流程——最糟糕的结果。然而，一个不相关的安全方面的任务使我们摆脱了这个陷阱。当时，许多分布式自动化都依赖于SSH。从安全的角度来看，这是很笨拙的，因为人们必须拥有机器的root权限才能运行大多数的命令。越来越多的人意识到，先进、持续的安全威胁迫使我们将SRE的特权降到最低，仅仅满足完成其工作所需的最低的特权。我们不得不用一个支持认证、ACL驱动，以及基于RPC的本地管理进程来取代sshd，这被称为Admin服务器，它拥有本地执行更改权限。这样一来，没有人可以绕过审计跟踪来安装或是修改服务器。对Admin服务器代码和Package仓库的修改通过代码评审来把关，这使得越权操作非常困难；赋予别人安装软件包的权限不会允许他们查看日志。Admin服务器会记录RPC请求者、全部参数，以及所有的RPC的结果，以提高调试和安全审计功能。

以服务为导向的集群上线流程

在下一次迭代中，Admin 服务器成为服务团队的工作流程的一部分，这包括作为机器管理的Admin 服务器（安装包和重新启动服务器），以及集群级别的Admin 服务器（如进入排水模式，或者服务上线）。SRE从在自己的主目录里维护shell脚本迁移到了构建评审过的RPC服务器与细粒度的ACL上。

后来，在认识到上线流程必须由服务团队维护之后，我们找到了一种以服务为导向的架构（SOA）来解决集群上线问题的方法：服务拥有者将负责创建一个Admin 服务器，处理系统在集群就绪之后发出的集群上线/下线RPC。反过来，每个团队将按照合同（API）提供自动上线所需的自动化，然而仍然可以随意改变底层实现细节。随着一个集群进入“网络就绪”，自动化系统将给每个负责上线的Admin 服务器发送RPC。

我们现在拥有的是低延迟、能力强以及非常精确的流程；更重要的是，这个流程在变更率、团队的数量，以及服务的数量每年翻一倍的情况下仍然保持可靠。

如前所述，集群上线自动化进化遵循这样一个路径：

1.操作人员触发手动操作（无自动化）。

2.操作人员编写，系统特定的自动化。

3.外部维护的通用自动化。

4.内部维护，系统特定的自动化。

5.不需要人为干预的自治系统。

虽然这种演变宽泛地说是成功的，但Borg的案例研究描述了我们考虑自动化问题的另一种方式。

Borg：仓库规模计算机的诞生

理解我们对自动化的态度的演变，以及何时何地部署自动化是最佳的另一种方式，就是考虑我们的集群管理系统发展的历史。[18]正如前文“MySQL On Borg”体现了手动操作到自动化的成功转变，集群上线流程则体现了不仔细考虑在哪里以及如何实现自动化所带来的不足。集群管理的研发也展示了另一个关于如何实行自动化的教训。和我们之前提过的两个例子一样，一个复杂的东西诞生于简单初始环境的不断演变。

Google的集群最初与其他人当时的小型网络部署方式很类似：具有特定用途和异构配置的很多机柜服务器。工程师将登录到一些知名的“主”机来执行管理任务；“黄金”二进制文件和配置保存在这些主机上。当时我们只有一个托管供应商，大多数的命名逻辑隐含地假定了位置信息。随着生产环境的增长，我们开始使用多个集群，多个域（集群名称）也变得有必要了。现在就需要一个文件来描述每个机器做了什么，以一些松散命名策略将机器编组。这个描述文件，加上一个并行化的SSH，允许我们一次重新启动（例如）所有的搜索机器。在当时，收到类似“搜索团队已经用完x1这台机器，爬虫团队现在可以使用了”的工单是很普通的。

自动化研发开始了。最初的自动化包括简单的Python脚本，执行如下操作。

- 服务管理：保障服务运行（例如，在段错误后重新启动）。
- 跟踪哪些服务应该运行在哪些机器上。
- 日志消息解析：SSH进入每一台机器，用正则表达式过滤日志。

自动化最终成长为使用一个数据库来跟踪机器状态，并且采用了更先进的监控工具。随着自动化系统的成长，我们现在可以自动管理机器的大部分生命周期：当机器坏了，移除服务，送去修理并且在它们修好后恢复配置。

但是退一步说，这种自动化是有用的，但是限制非常多，因为该系统的抽象对象与物理机紧密相关。我们需要一个新的方法，因此Borg（参见文献[ver15]）诞生了：从相对静态的主机/端口/作业分配，到将一系列机器看作受管理的海量资源的创新。Borg成功的核心—以及它的核心理念—是把集群管理变成了一个可以发送API的中央协调主体。这从另外一个维度解放了效率、灵活性以及可靠性：对比以前的机器的“独占”模式，Borg能够让机器来调度任务，例如在同一台机器上同时进行批处理和面向用户的任务。

此功能最终使得连续性的、自动的操作系统升级只需要很少的持续[19]工作—这种工作不会根据生产部署的总规模而增加。当机器状态略有偏差时，可以自动进行修补；对SRE来说，机器的损坏和生命周期管理基本上已经不需要任何操作了。成千上万的机器加入系统，或者出现问题，被修复，这一切都不需要SRE的任何操作。回应 Ben Treynor Sloss 曾说过的话：通过将问题看作是一个软件问题的方法，最初的自动化努力给我们争取了足够的时间，把集群管理转化为自治系统，而不是自动化系统。我们通过相关的数据分布、API、集中式架构，以及经典的分布式软件系统研发进入了基础设施管理领域。

这里有一个有趣的比喻：我们可以将单机环境与集群管理抽象的发展之间直接映射。用这个方法，在另一台机器上重新调度某个程序与进程从一个CPU移动到另一个CPU的过程十分相似：当然，这些计算资源其实是在网络链接的另一端，但是这又有多重要呢？从这个视角出发，重新调度看起来像是系统的一个固有特征，而不是可以“自动化”的—不

管怎样，人类的反应速度肯定是不够快。在集群上线这个例子中也类似：在这个比喻中，集群上线仅仅是增加额外的调度能力，有点像在一台电脑上添加磁盘或内存。然而，一个单一节点的计算机在大量组件出现问题时，是不会继续运行的。一旦全球计算机的增长超过了一定规模时，它必须是自我修复的，因为根据统计学来说，每秒它都会发生大量故障。这意味着，随着系统的层次结构不断上升，从手动触发，到自动触发，到自主化，一些自我检查的能力是必需的。

可靠性是最基本的功能

当然，为了有效地进行故障调试，自我检查中所依赖的内部运作细节也应该暴露给管理整体系统的操作员。在非计算机领域中对自动化影响的类似讨论—例如，民航[20]或工业应用中—经常会指出高效的自动化的缺点[21]：随着时间的推移，操作员与系统的有用的、直接接触会逐渐减少，因为自动化会覆盖越来越多的日常活动。不可避免的，当自动化系统出现问题时，操作员将无法成功地操作该系统。

由于缺乏实践，他们已经丧失了反应的流畅性，他们有关系统“应该”做什么的心理模型不再反映现实中系统“正在进行”的活动[22]。这种情况在系统非自主运行时出现得更多，即，当自动化逐渐取代了手动操作，假设其他的手工操作仍然可能执行，并且如之前一样一直可用。令人难过的是，随着时间的推移，这一假设终将不再正确：这些手动操作最后将无法执行，因为允许它们执行的功能已经不存在了。

Google也经历过自动化在某些条件下是有害的情况，参看下面“自动化：允许大规模故障发生”补充材料。但是以Google的经验来看，在更多的系统中自动化和自主化的行为不再是可选择的附加项。随着服务规模扩大，肯定是这样的。但是不论系统规模大小，系统中具有更多自主行为的系统仍然有很多好处。可靠性是最基本的功能，并且自主性、弹性行为是达到这一特征的有效途径。

建议

读过本章的案例之后，你肯定会觉得在达到Google的规模之前不需要进行任何自动化。这是不正确的，有以下两个原因：自动化提供的不

仅仅是对时间的节省，所以在单纯的时间花费和时间节省的计算之外也值得实施。但是，最有效的方法其实在设计阶段：更快地交付和更快地迭代可能会帮助你更快地实现功能，但是却很难形成一个有弹性的系统。对足够大的系统来说，进行改造时加入自主行为是很难的，但软件工程的良好标准的做法将会有很大的帮助：解耦子系统，引入API，最大限度地减少副作用等。

自动化：允许大规模故障发生

Google运行十几个自有的大型数据中心，但是我们也依赖很多运行在第三方托管设施（或“colo”）内的机器。这些colo中的机器是用来终止大部分传入的连接的，或是作为自己的内容分发网络的缓存，以降低用户的等待时间。在任何一个时间点，都有很多机架被安装或被退役；这两个过程大部分都是自动化的。拆除中的一个步骤包括覆写机架上所有机器的磁盘的全部内容，同时一个独立的系统随后会验证机器被成功擦除。我们称这个过程为“磁盘擦除”（diskerase）。

很久以前，负责某个特定机架退役的自动化系统出现了问题，但只有磁盘擦除步骤已经成功完成。随后，清退过程被重启，以调试为什么失败。在这次操作中，当该程序试图给磁盘清除系统发送机架中的机器列表时，代码得出需要进行清空的机器列表是（正确的）空的。不幸的是，空列表有着特殊含义，它被解释为“所有”。这意味着自动化系统几乎将colo中的所有机器都送去进行磁盘清除了。

几分钟内，高效的磁盘清除程序擦除了我们的CDN上所有机器的磁盘，这些机器再也不能终止用户连接了（或做任何有用的事情）。我们仍然可以从我们自己的数据中心来服务全体用户，事实上几分钟后对外部的唯一影响就是会有轻微的延迟增加。据我们所知，由于良好的容量规划，很少有用户注意到这个问题（至少我们这一点做得不错！）。接下来，我们花费了大概两天时间重装受影响的colo机架上的机器；然后我们又将接下来的几周时间用来进行代码审计，在我们的自动化系统中添加更多的合理性检查，包括速率限制，以及使整个退役流程具有幂等性。

第8章 发布工程

作者：**Dinah McNutt**

编辑：**Betsy Beyer、Tim Harvey**

发布工程（Release Engineering）是软件工程内部一个较新、发展较快的学科。简单来说，这个学科专注于构建和交付软件（参见文献[McN14a]）。发布工程师通常对源代码管理、编译器、构建配置语言、自动化构建工具、包管理器和安装器等非常了解（甚至是这方面的专家）。他们的技能横跨很多领域：开发、配置管理、测试集成、系统管理，甚至用户支持。

为保障服务可靠运行需要可靠的发布流程。SRE需要保证二进制文件和配置文件是以一种可重现的、自动化的方式构建出来的。这样每一次发布才是可以重复的，而不是“独特的雪花”（俚语，意指没有两片雪花是完全相同的）。对发布流程的任何改变都应该是有意为之，而不是意外之举。SRE关注从源代码到部署的整个流程。

发布工程是Google内部的一项具体工作。发布工程与产品研发部门的软件工程师（SWE），以及SRE一起定义发布软件过程中的全部步骤—包括软件是如何存储于源代码仓库中的，构建时是如何执行编译的，如何测试、打包，最终进行部署的。

发布工程师的角色

Google是一个数据驱动的公司，发布工程也不例外。我们有各种各样的工具提供各种各样的数据。例如，从代码修改提交到部署到生产环境一共需要多长时间（也就是发布速度），又比如统计构建配置文件中某个特性的使用率（参见文献[Ada15]）。大部分这些工具都是由发布工程师设计和开发的。

发布工程师利用这些工具定义一些最佳实践，来保障软件项目可以一致地、可重复地进行发布。我们的最佳实践覆盖整个发布过程中的所有元素。例如，编译器功能开关、编译结果中的版本编号的格式、构建

过程中必须执行的步骤等。确保我们的工具在默认情况下就能正确工作，并且有合理的文档作为辅助，可以让开发团队专注于功能和用户，而不需要花费时间重新发明软件发布的轮子（经常还是不圆的轮子）。

Google有很多SRE负责产品更新的安全部署过程，保障这些服务可以正常运行。为了保障软件发布流程能够满足业务需求，发布工程师与SRE紧密协作，为变更的测试进行无缝发布，以及为变更的顺利回滚等制定策略。

发布工程哲学

发布工程师的日常工作是由下列4个主要的工程与服务哲学指导的。

自服务模型

为了应对大规模扩张，每个团队必须能够自给自足。发布工程师开发工具，制定最佳实践，以便让产品研发团队可以自己掌控和执行自己的发布流程。虽然我们有几千个工程师和几百个产品，Google仍然能够以很快的速度发布新产品。这是因为每一个团队都可以决定多久或者什么时候来发布产品的新版本。发布过程可以自动化到基本不需要工程师干预，很多项目都是利用我们的自动构建工具和部署工具自动构建、自动发布的。发布过程是真正的自动化的，工程师仅仅在发生问题时才会进行干预。

追求速度

面向用户的软件组件（例如Google搜索服务的很多组件）重新构建非常频繁，因为我们的目标是让用户可见的功能越快上线越好。我们同时也认为频繁的发布可以使得每个版本之间的变更减少。这种方式使得测试和调试变得更简单。有些团队每小时构建一次，然后在所有可用的构建版本中选择某个版本进行发布。选择过程是基于测试结果与所包含的功能列表共同得出的。还有的团队采用一种“测试通过即发布”（Push On Green）的发布模型，也就是说，部署每个通过所有测试的版本（参见文献[Kle14]）。

密闭性

构建工具必须确保一致性和可重复性。如果两个工程师试图在两台不同的机器上基于同一个源代码版本构建同一个产品，构建结果应该是相同的。[23]我们的构建过程都是密闭的（hermetic），意味着它们不受构建机器上安装的第三方类库或者其他软件工具所影响。构建过程使用指定版本的构建工具（编译器），同时使用指定版本的依赖库（第三方类库）。编译过程是自包含的，不依赖于编译环境之外的其他服务。

当修复某个运行在生产环境中的软件的Bug，而需要重新构建之前的一个发布版本时，一般来说是比较复杂的。Google按照之前的源代码版本，加入一些后来提交的改动来构建新的发布来解决这个问题，这种方式被称为摘樱桃（cherry picking）。构建工具自身也是放置在与被构建的程序同一个源代码仓库之中的。因此，如果要对上个月构建的某个项目进行cherry picking，仍然会用当时的编译器版本，而不会用到这个月新的编译器版本，这样可以保证编译器功能的一致性。

强调策略和流程

多层安全和访问控制机制可以确保在发布过程中只有指定的人才能执行指定的操作。我们主要关注的操作有如下几项：

- 批准源代码改动—通过源代码仓库中的配置文件决定。
- 指定发布流程中需要执行的具体动作。
- 创建新的发布版本。
- 批准初始的集成请求（也就是一个以某个源代码仓库版本为基础的构建请求），以及后续的cherry picking请求。
- 实际部署某个发布版本。
- 修改某个项目的构建配置文件。

几乎所有对源代码的修改都需要进行代码评审，这与我们日常开发工作流程是完美结合的。我们的自动化发布系统可以提供每个发布中包

含的所有改动的报告，与其他的构建结果一起归档。SRE可以了解每个新发布中包含的具体改动，在发布出现问题时可以更快地进行在线调试。

持续构建与部署

Google开发了一个自动化的发布系统：Rapid。该系统利用一系列Google内部技术执行可扩展的、密闭的，以及可靠的发布流程。下面几小节描述了Google内部的软件生命周期，以及我们是如何利用Rapid和其他相关工具管理这种周期的。

构建

Blaze[24]是Google的构建工具，它支持多种编程语言，如Google内部标准的 C++、Java、Python、Go 以及JavaScript。工程师利用Blaze定义构建目标，即构建的输出结果，例如Jar文件，同时给每个目标指定依赖关系。当进行具体构建时，Blaze会自动构建目标的全部依赖。

构建目标（二进制文件，以及对应的测试等）定义在Rapid的项目配置文件中。某个项目特有的功能开关，例如一些特有的构建标识符等，会由Rapid传递给Blaze。所有二进制文件都支持用一个命令显示自身的构建时间、构建源代码版本，以及构建标识符，这样我们就可以很容易地将一个二进制文件与构建过程对应起来。

分支

所有的代码都默认提交到主分支上（mainline）。然而，大部分的项目都不会直接从主分支上进行直接发布。我们会基于主分支的某一个版本创建新分支，新分支的内容永远不会再合并入主分支。Bug修复先提交到主分支，再cherry picking到发布分支上。这种方式可以避免在第一次构建之后，再引入主分支上的其他的无关改动。利用这种分支与cherry picking的方法，可以明确每个发布版本中包含的全部改动。

测试

一个持续测试系统会在每个主分支改动提交之后运行单元测试，这样我们可以快速检测构建错误和测试错误。建议使用项目中定义的构建目标及测试目标的执行结果来决定是否发布某个版本。同时建议使用最后一个测试全部通过的软件版本来进行最新的发布。这些方法可以降低在真正发布时由于主分支上其他无关改动造成问题的几率。

在发布过程中，我们会使用该发布分支重新运行全部单元测试，同时为测试结果创建审核记录。这一个步骤非常重要，因为如果一个发布过程需要cherry picking，发布分支可能会包含主分支上不存在的一个代码版本。我们必须确保在发布分支上全部测试确实通过。

为实现持续测试系统，我们使用一套独立的测试环境来在打包好的构建结果上运行一些系统级别测试。这些测试可以从Rapid网站上手工启动。

打包

软件通过Midas Package Manager（MPM）系统分发到生产机器上。MPM基于Blaze规则中列出的构建结果和权限信息构建MPM包。每个包有固定名称（如search/shakespeare/frontend），记录构建结果的哈希值，并且会加入签名以确保真实完整性。MPM同时支持给某个版本的包打标签。Rapid也会加入一个构建ID标签，这样某个包可以用名字和这个标签来唯一识别。

我们可以给某个MPM包加标签，标记该MPM包在整个发布过程中的位置（如dev、canary或production等）。如果将某个现有标签应用到新包上，这个标签会自动从原来的包上移除。例如，如果一个包标记为canary，之前的canary包上的标签就被自动去掉了，后续安装canary版本的包的人会自动使用最新的包版本。

Rapid系统

图8-1　展示了Rapid系统中的主要组件。Rapid是用Blueprint文件配置的。Blueprint文件是一种利用Google内部配置语言写成的，用来定义构建目标和测试目标、部署规则，以及一些管理用信息（例如项目负责

人信息）。基于角色的访问控制列表可以决定谁能执行哪些动作。

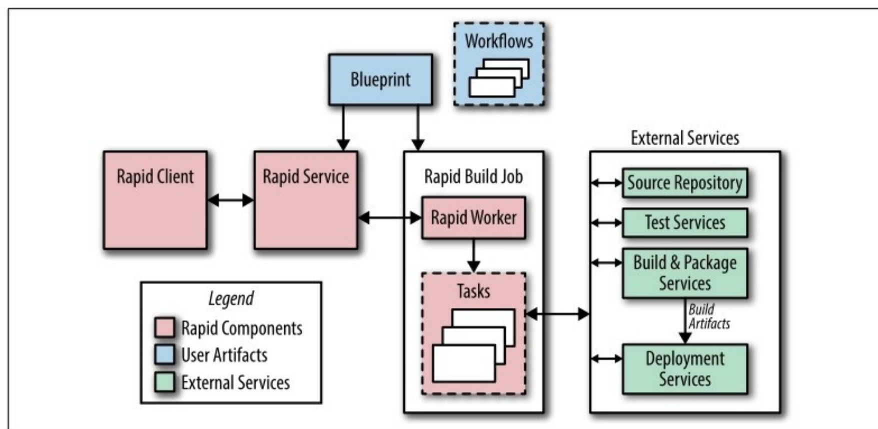


图8-1：简化版Rapid架构，展示了系统的主要组件。

每个Rapid项目都有一些工作流，定义了发布流程中的具体动作。工作流可以线性或者并发执行，某个工作流也可以启动另外一个工作流。Rapid将工作请求分发给运行在Borg系统上的生产服务器。因为Rapid使用Google的生产基础设施，我们可以同时处理几千个发布请求。

典型的发布流程按如下顺序进行：

1.Rapid使用集成版本号（通常自动从持续测试系统获取）创建新的发布分支。

2.Rapid利用Blaze编译所有的二进制文件，同时执行所有的单元测试，这两个过程通常是并发进行的。编译和测试各自在独立的环境中进行，而非Rapid工作流运行的环境中。这种隔离使得并发更容易一些。

3.构建结果随后可以用来运行系统级集成测试，同时进行测试部署。典型的测试部署过程是在系统测试完成之后，在生产环境中启动一系列Borg任务。

4.每一步的结果都有日志记录。另外产生一份与上次发布对比包含的所有新的改动列表的报告。

Rapid可以管理发布分支与cheery picking。每个具体的cherry picking请求可以被单独批准和拒绝。

部署

Rapid经常被用来直接驱动简单的部署流程。它可以根据Blueprint文件定义的部署规则，利用具体的任务执行器（executor）来用新构建的MPM包更新Borg任务。

我们使用Sisyphus,SRE开发的一个通用的发布自动化框架，来执行更为复杂的部署任务。一个发布（rollout）是由一个或多个任务组成的一个逻辑工作单元。Sisyphus提供了一系列可扩展的Python类，以支持任意部署流程。同时，它还有一个监控台，可以用来详细控制每个发布的执行，以及监控发布流程。

在典型的集成流程中，Rapid在某个Sisyphus系统中创建一个新的发布。Rapid知道自己构建的MPM包的build标签，可以在创建发布时指定这个标签。Sisyphus可以利用这个build标签来指定究竟使用哪个MPM版本进行部署。

Sisyphus，可以支持简单的发布流程，也可以支持复杂的发布流程。例如，我们可以立即更新所有的相关任务，也可以在几个小时的周期内，一个接一个地更新集群版本。

我们的目标是让部署流程与服务的风险承受能力相结合。在开发环境或者预生产环境中，我们可能会每小时构建一次，同时在所有测试通过之后自动发布更新。对于大型面向用户的服务来说，我们可能会先更新一个集群，再以指数速度更新其他集群直到全部完成。对敏感的基础设施服务来说，我们可能会将发布扩展到几天内完成，根据这些实例所在的地理位置交替进行。

配置管理

配置管理是发布工程师与SRE紧密合作的一个区域。虽然初看起来，配置管理可能很简单，但是这其实是不稳定性的一个重要来源。因此，我们的发布流程和系统运维与配置管理流程都随着时间不停地发

展。今天我们使用下面几段描述的模型来分发配置文件。所有这些模型都需要将配置文件存放于我们的主要代码仓库中，同时进行严格的代码评审。

使用主分支版本配置文件。这是配置Borg服务的第一个方法（以及配置Borg之前的那些系统）。使用这个模型，开发者和SRE可以同时修改主分支上的配置文件。这些修改经过代码评审之后会应用到正在运行的系统上。这样做的结果是，二进制文件的发布与配置文件的修改是异步进行的。虽然理解起来和执行起来都比较容易，但这种方式经常会造成提交的版本和实际运行的配置文件不一致，因为任务必须要经过更新才能应用这些变更。

将配置文件与二进制文件打包在同一个MPM包中。对没有多少配置文件的项目来说，或者那些每次发布都会改变文件的项目来说，配置文件可以直接和二进制文件放在一个MPM包里。虽然这个策略在灵活性上有一定限制，但是简化了部署，仅仅需要安装一个包。

将配置文件打包成MPM配置文件包。我们也可以将密闭原则应用到配置管理上。二进制配置文件一般与某个二进制版本紧密相关，我们也可以利用编译系统和打包系统来发布配置文件。就像二进制文件那样，可以利用构建ID来重新构建某一时刻的配置文件。

例如，某个实现了新功能的变更可以与配置该功能的配置文件一起发布。通过打包两个MPM包，一个二进制文件，一个配置文件，可以对两个包进行单独修改。如果某个功能需要一个功能开关☆rst_folio，但是我们后面发现这个开关值应该为bad_quarto（这些是没有实际意义的名字），可以将这个变更cherry picking到发布分支上，重新构建配置包，再实际部署。这种方式可以避免再构建一次二进制文件。

我们可以利用MPM的标签功能选择哪些MPM包应该同时安装。much_ado这个标签可以应用到上面那段中的MPM包上，这样我们可以用一个标签同时获取两个版本。因为这些标签在每个MPM命名空间内部都是唯一的，只有最后一个包才能被用到。

从外部存储服务中读取配置文件。某些项目的配置文件需要经常改

变，或者动态改变（在二进制文件运行时）。这些文件可以存放在Chubby、Bigtable或者Google自己的基于源代码仓库的文件系统中（参见文献[Kem11]）。

总之，项目负责人在分发和管理配置文件时有多种选择，可以按需决定究竟哪种最适合该服务。

小结

虽然本章讨论的是Google的发布工程的特有方法，以及发布工程师与SRE共同合作的区域，但其实这些实践适用于更广阔的范围。

不仅仅只对Google有用

当采用合适的工具、合理的自动化方式，以及合理的政策时，开发团队和SRE都无须担心如何发布软件。发布过程可以像按一个按钮那么简单。

大部分公司，不论团队大小和使用何种工具，都面临着同样的发布工程问题：如何管理包的版本？应该采用持续构建和部署的模型，还是应该定期构建？发布的频率应该怎样？应该使用什么策略管理配置文件？哪些发布过程的指标比较有用？

Google 发布工程师开发自己工具的原因是因为第三方供应商提供的工具无法适应我们的海量规模。自定义过程使得我们可以在工具中加入对发布流程政策的支持（甚至是对这些政策的强制执行）。然而，这些政策必须先被正确定义，这样才能指导工具功能的创造。不管最终的政策是否需要自动化，或者是否需要强制执行，任何组织都应该先花一些时间定义自己的发布政策。

一开始就进行发布工程

发布工程经常是“事后诸葛亮”，随着平台和服务的规模与复杂度不断增加，这种理念一定需要改变。

团队应该在开发流程开始时就留出一定资源进行发布工程工作。尽

早采用最佳实践和最佳流程可以降低成本，以免未来重新改动这些系统。

开发团队、SRE和发布工程师的紧密协作是很重要的。发布工程师需要明白代码开发时对构建与部署的预期。开发团队不应该只是编写代码，然后“将结果扔过墙”，两个团队必须互相了解。

每个具体的项目团队需要决定何时进行发布工程。因为发布工程是一个相对来说较新的学科，管理层不一定会在项目早期为此进行计划和提供资源。因此，当应用发布工程最佳实践时，一定要考虑到它在整个产品生命周期中的地位，尤其是在项目早期。

更多信息

关于其他发布工程的信息，请参考以下演讲，每个都有对应的视频。

● How Embracing Continuous Release Reduced Change Complexity（http://usenix.org/conference/ures14west/summit-program/presentation/dickson）,USENIX Release Engineering Summit West 2014,[Dic14]

● Maintaining Consistency in a Massively Parallel Environment（https://www.usenix.org/conference/ucms13/summit-program/presentation/mcnutt）,USENIX Configuration Management Summit 2013,[McN13]

● The 10 Commandments of Release Engineering（https://www.youtube.com/watch? v=RNMjYV_UsQ8）,2nd International Workshop on Release Engineering 2014,[McN14b]

● Distributing Software in a Massively Parallel Environment（https://www.usenix.org/conference/lisa14/conference-program/presentation/mcnutt）,LISA 2014,[McN14c]

第9章 简单化

作者：**Max Luebbe**

编辑：**Tim Harvey**

可靠性只有靠对最大程度的简化不断追求而得到。

——C.A.R.Hoare,Turing Award lecture

软件系统本质上是动态的和不稳定的。[25]只有真空中的软件系统才是永远稳定的。如果我们不再修改代码，就不会引入新的Bug。如果底层硬件或类库永远不变，这些组件也就不会引入Bug。如果冻结当前用户群，我们将永远不必扩展系统。事实上，一个对SRE管理系统的方法不错的总结是：“我们的工作最终是在系统的灵活性和稳定性上维持平衡。”[26]

系统的稳定性与灵活性

有的时候为了灵活性而牺牲稳定性是有意义的。我在面临一个不熟悉的问题域时，经常进行“探索性编码”—给我写的任何代码设置一个明确的保质期，我清楚地知道自己需要先探索以及失败才能真正理解需要完成的任务。这种带保质期的可以在测试覆盖和发行管理上更宽松，因为它永远不会被发布到生产环境或被用户使用。

对于大多数生产环境软件系统来说，我们想要在稳定性和灵活性上保持平衡。SRE通过创造流程、实践以及工具，来提高软件的可靠性。同时，SRE需要最小化这些工作对于开发人员的灵活性造成的影响。事实上，SRE的经验表明，可靠的流程会提高研发人员的灵活性：快速、可靠的产品发布使得生产系统中的变化显而易见。这样，一旦出现错误，找到和改正错误的时间会更少。在开发过程中引入可靠性可以让开发人员关注那些真正需要关注的事情—软件和系统的功能与性能。

乏味是一种美德

与生活中的其他东西不同，对于软件而言，“乏味”实际上是非常正面的态度！我们不想要自发性的和有趣的程序；我们希望这些程序按设计执行，可以预见性地完成商业目标。Google工程师Robert Muth曾说过，“与侦探小说不同，缺少刺激、悬念和困惑是源代码的理想特性。”生产环境中的意外是SRE最大的敌人。

Fred Brooks 在他写的名为No Silver Bullet的文章（参见文献[Bro95]）中表示，关注必要复杂度和意外复杂度之间的区别非常关键。必要复杂度是一个给定的情况所固有的复杂度，不能从该问题的定义中移除，而意外复杂度则是不固定的，可以通过工程上的努力来解决。例如，编写一个Web服务器需要处理快速提供Web页面的必要复杂度。但是，如果我们用Java编写该服务器，试图减少GC的影响就可能会引入意外复杂度。

为了最小化意外复杂度，SRE团队应该：

- 在他们所负责的系统中引入意外复杂度时，及时提出抗议。
- 不断地努力消除正在接手的和已经负责运维的系统的复杂度。

我绝对不放弃我的代码

因为工程师也是人，他们经常对于自己编写的代码形成一种情感依附，这些冲突在大规模清理源代码树的时候并不少见。一些人可能会提出抗议，“如果我们以后需要这个代码怎么办？”“我们为什么只是把这些代码注释掉，这样稍后再使用它的时候会更容易吗？”“为什么不增加一个功能开关？”这些都是糟糕的建议。源代码控制系统中的更改反转很容易，数百行的注释代码则会造成干扰和混乱（尤其是当源文件继续演进时）；那些由于功能开关没有启用而没有被执行的代码，就像一个定时炸弹一样等待爆炸，正如Knight Capital的痛苦经历（参考“Order In the Matter of Knight Capital Americas LLC”，文献[sec13]）。

极端地说，当你指望一个Web服务7×24可用时，在某种程度上，每一行新代码都是负担。SRE推崇保证所有的代码都有必须存在的目的的实践。例如，审查代码以确保它确实符合商业目标，定期删除无用代

码，并且在各级测试中增加代码膨胀检测。

“负代码行”作为一个指标

术语“软件膨胀”用来描述软件随着时间的推移不停地增加新功能而变得更慢和更大的趋势。臃肿的软件直观上来看就是不可取的，从SRE的视角中可以更清晰地描述这种情况的消极方面：添加到项目中的每行代码都可能引入新的缺陷和错误。较小的项目容易理解，也更容易测试，而且通常缺陷也少。从这一观点出发，当我们感觉到增加新功能需求时，应该保持保守的态度。我曾经做过的一些最令人满意的编码工作就是删除了数千行已经没用的代码。

最小 API

法国诗人 Antoine de Saint Exupery 曾写道，“不是在不能添加更多的时候，而是没有什么可以去掉的时候，才能达到完美。”（参见文献[Sai39]）这个原则同样适用于软件的设计和构建。API是这个规则应该遵循的一个清晰的例子。

书写一个明确的、最小的API 是管理软件系统管理简单性必要的部分。我们向API消费者提供的方法和参数越少，这些API就越容易理解，我们就能用更多的精力去尽可能地完善这些方法。同时，一个反复出现的主题是：有意识地不解决某些问题可以让我们能够更专注核心问题，使得我们已有的解决方案更好。在软件工程上，少就是多！一个很小的，很简单的API通常也是一个对问题深刻理解的标志。

模块化

在API与单个二进制文件以外，适用于面向对象编程的许多经验法则也适用于分布式系统的设计。对系统中某个部分进行隔离式的变更的能力对创建一个可以运维的系统来说非常必要。具体而言，在二进制文件之间或者二进制文件与配置之间推行松耦合，是一种同时提高开发人员的灵活性和系统的稳定性的简化模式。如果在一个更大系统的某个组件中发现一个错误，我们可以修复这个错误并且独立于系统的其他部分更新该程序。

虽然API提供的模块化可能看上去很容易理解，但是如何将模块化的概念延伸到API的变更就没那么明显了。某个API的一个变更就可以迫使开发人员重建他们的整个系统，同时承担引入新问题的风险。通过将API版本化，可以允许开发人员继续使用它们的系统所依赖的版本，以更安全和深思熟虑的方法升级到新的版本。这样不必要求整个系统的每一次功能增加或改进都需要全面的生产更新，整个系统中的更新节奏可以不同。

随着系统变得越来越复杂，API与二进制文件之间的责任分离变得越来越重要。某种程度上，这与面向对象编程中的类设计类似：正如普遍认同的，编写一个其中包含无关功能的“大杂烩”类是一个糟糕的实践。构建和发布“util”或“misc”二进制文件同样也是个糟糕的实践。一个设计良好的分布式系统是由一系列合作者组成的，每一个合作者都具有明确的、良好定义的范围。

模块化的概念同样适用于数据格式。Google的Protobuf[27]的一个主要优势和设计目标就是创建一个同时向后和向前兼容的传输格式。

发布的简单化

简单的发布流程总的来说要比复杂的发布流程更好。测量和理解单一变化的影响要比同时应对一系列变化更加容易。如果同时发布100个不相关的系统更改，而系统性能变差了，我们需要花费大量时间和努力来定位哪些改变影响了系统性能，以及它们是怎样影响的。如果发布是按更小的批次进行的，我们就可以更有信心地进行更快的发布，因为每个变更在系统中的影响可以独立理解。这种发布方式跟机器学习中的梯度下降法类似。我们通过每次进展一点，同时考虑每次改变对系统的改善和退化来寻找最佳方案。

小结

这一章反复重申的主题是：软件的简单性是可靠性的前提条件。当我们考虑如何简化一个给定的任务的每一步时，我们并不是在偷懒。相反，我们是在明确实际上要完成的任务是什么，以及如何更容易地做到。我们对新功能说“不”的时候，不是在限制创新，是在保持环境整

洁，以免分心。这样我们可以持续关注创新，并且可以进行真正的工程工作。

[1]本节的一个早期版本出现在；login：的文章中（2015年8月，第40期，第4章）。

[2]被称为“bang/bang”控制，更多信息请参见https://en.wikipedia.org/wiki/Bang–bang control。

[3]在讨论SLA的大部分时候，实际上是在讨论SLO。如果某个人说道“违反SLA”，实际是指没有达到SLO，真实的“违反SLA”可能会触发一次违反合约的法律诉讼！

[4]如果SRE团队无法说服研发团队接受任何一个SLO，那么这个产品可能压根不需要SRE团队的支持。

[5]Failure Injection（参见文献[Ben12]）是另外一个方法，也可以用来调节用户的预期。

[6]我们使用Andy Grove 在 Intel首创的 Objectives and Key Results系统。

[7]Google 员工每周要写一个短小的无固定格式的工作总结，称为“Snippets”。

[8]我们在讨论一个任务“不是琐事，因为它需要主观判断”的时候要非常谨慎。需要仔细考虑这项任务是否本质上需要主观判断，并且不能通过更好的设计来消除这种需要。比如，某项服务可能每天都要发出数次警报，并且每个警报都需要大量主观判断的复杂响应。这样的服务从设计上来说就是不佳的，很多复杂性是不必要的。该系统需要被简化、重构以消除掉潜在的故障情况，或者自动处理这些情况。重新设计和重构完成以及完整发布之前，依赖主观判断来应对每一个警报的工作就绝对是琐事。

[9]有时候也称为“垃圾警报”，因为很少有人会去关注他们，这些警报也不会触发任何操作。

[10]如果1%的请求需要10倍的平均时间，这就意味着其他请求大

概只需要平均值的一半。除非有统计延迟的分布情况证明，大多数请求都接近平均值这个假设一般是不成立的。

[11]关于警报过多的“狼来了”效应，请参见Applying Cardiac Alarm Management Techniques to Your OnCall（文献[Hol14]）。

[12]零冗余（N+0）的情况也应该算作是即将发生的故障的情况，同样，接近满载的情况也一样！关于冗余度的讨论请参见https://en.wikipedia.org/wiki/N%2B1_redundancy。

[13]对于那些已经明确地理解自动化价值的读者，可以直接跳到本章后面的“自动化对Google SRE 的价值”一节。然而，请注意，我们的描述包含一些阅读本章其他部分可能有用的细微差别。

[14]在构建自动化系统的过程中获得的专业知识本身也有价值；工程师可以在构建自动化的过程中更深刻地理解现有流程，之后可以更快速地自动化新的流程。

[15]可观看XKCD动画：http://xkcd.com/1205/。

[16]参考例子，http://blog.engineyard.com/2014/pets-vs-cattle。

[17]当然，不是每个需要管理的系统都提供了可调用管理API——这要求我们必须采用某种工具，例如执行CLI，或者执行自动化的网站单击。

[18]我们已经压缩和简化了这段历史来帮助理解。

[19]作为一个很小、保持不变的数字。

[20]具体细节请参看https://en.wikipedia.org/wiki/Air_France_Flight_447。

[21]具体细节请参看文献[Bai83]和[Sar97]。

[22]这是定期演习的另外一个不错的理由；参看第28章的“故障处理分角色演习”一节。

[23]Google的全部源代码存放于一个单独的代码仓库中（参见文献[Pot16]）。

[24]Blaze的开源版本为Bazel，可参见网站上的“Bazel Faq”,http://bazel.io/faq.html。

[25]一般来说，复杂系统都是这样的，可参见文献[Per99]和[Coo00]。

[26]这句话是由我的前任经理 Johan Anderson创造的，当时我刚刚加入SRE。

[27]Protocol Buffer，也被称为“protobuf”，是一个语言中性、平台中立、用于序列化结构化数据的可扩展方法。更多细节请参考https://developers.google.com/protocol-buffers/docs/overview#a-bit-of-history。

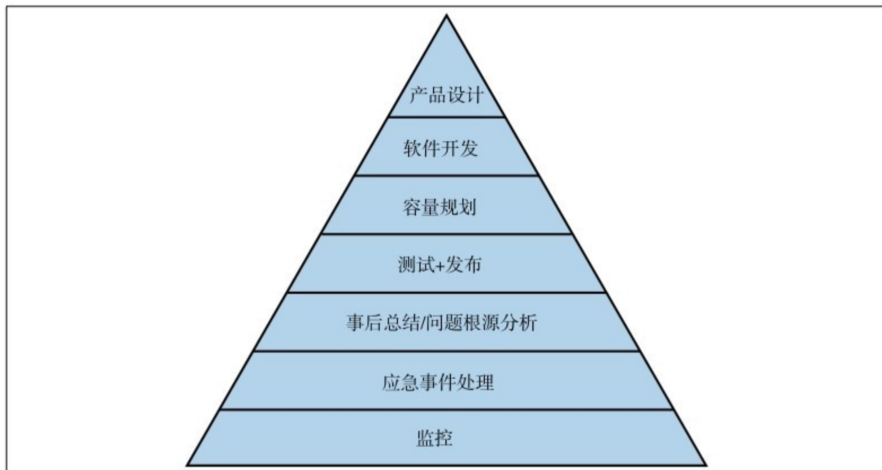
第III部分 具体实践

简单来说，SRE的职责是运维一个服务。该服务由一些相关的系统组件组成，为最终用户提供服务（可以是内部用户或外部用户）。SRE的终极责任是确保该服务可以正常运转。为达成这个目标，SRE需要完成以下一系列工作：开发监控系统，规划容量，处理紧急事件，确保事故根源被跟踪修复等。这一部分将主要讨论SRE维护大型分布式计算系统的指导理念和最佳实践。

Abraham Maslow 曾经将人的生活需求分类论述（参见文献[Mas43]）。鉴于此，我们可以将一个服务的健康程度指标分为低级需求：能够正常对外提供服务，和高级需求：SRE能够主动控制服务状态，而不是被动救火。这个理念从多年实践中积累得来，却一直没有被详细定义过。

2013年年末至2014年年初，Google SRE,Mikey Dickerson，[1]临时被抽调加入一个由美国政府组建的小组，负责帮助解决healthcare.gov线上遇到的问题。在尝试向来自其他背景的人解释Google SRE是如何看待服务可靠性问题时，这个理念才第一次被详细定义下来。

我们用图III-1中的层级模型来详细论述一下服务可靠度指标的基本需求和高级需求。



图III-1：服务可靠度层级模型

监控

离开了监控系统，我们就没有能力辨别一个服务是不是在正常提供服务。没有一套设计周全的监控体系就如同蒙着眼睛狂奔。作为一个合格的系统运维人员，我们需要在用户之前发现系统中存在的问题。在第10章将讨论有关警报系统的设计哲学和工具。

应急事件处理

SRE并不是为了on-call值班而值班，on-call 值班只是我们实现服务目标的一种工具。经常参与轮值有助于每一个SRE了解和熟悉大型分布式计算系统的失败模式。如果我们可以找到一种方式使得值班不再必要，SRE肯定会第一时间采用。在第11章中，将详细解释我们是如何平衡on-call轮值与其他工作的。

一旦SRE发现了系统中存在的问题，要如何解决呢？正确的解决方案不一定是当场把问题一次性修复好，而可以靠降低系统准确度、关闭一些不重要的功能，或者将用户流量导向其他没有问题的任务实例等手段暂时缓解问题。解决方案的细节肯定是和每个服务和团队相关的。但是如何有效地应对紧急问题的方法论是每个团队都适用的。

在第12章中，我们提供了一套结构化方案来解决问题的第一步：找到问题所在。

在应急事件处理中，由于压力很大，很多情况下人们急于解决问题，会绕开必要的流程。在第13章和第14章中，我们讨论了如何通过合理的流程有效地减小事故的影响范围，以缓解整个部门由生产事故带来的压力。

事后总结和问题根源分析

我们的目标是接收警报，然后人工解决新的、有挑战性的服务问题。反复不断地修复同样的问题简直太无聊了。事实上，这个理念是SRE和传统运维团队理念中最大的不同点。下面两章详细描述了这个主题。

第15章，描述了如何建立起“无指责”、“对事不对人”的团队文化，只有做到这一步，才能有效了解到在一次事故中哪些地方出现了问题（以及哪些地方做得非常好）。

第16章，在这个基础上，我们简要描述了一个内部工具，事故跟踪系统。SRE团队使用该系统跟踪最近发生的生产事故、原因以及解决的具体过程。

测试

当我们发现经常出现问题的组件或者流程时，下一步就是如何避免它再次发生故障。通过增加测试，保证软件在发布到生产环境中时不会出现某些类型的问题。在第17章中详细描述了相关的最佳实践。

容量规划

在第18章中，我们给出了一个具体的工具开发实践案例。SRE 利用Google 内部工具Auxon，自动化进行服务容量规划。

容量规划后，我们还需要保证合理的负载均衡体制能够正确地使用这些服务容量。在第19章中，我们讨论了用户请求是如何发送到数据中心的。在第20章和第21章中，我们详细描述了数据中心中的负载均衡体系是保障服务可靠度的关键。

最后，在第22章中，我们讨论了如何从系统设计层面与战术层面应对“连锁故障”。

软件研发

Google SRE模型的要点之一就是SRE一半的精力都花在设计和开发大规模软件系统上。

在第23章中，我们解释了Google许多系统的核心组件—“分布式共识”系统（Paxos的新说法），例如全球分布式定时任务系统。在第24章中，我们描述了SRE是如何设计和部署一个跨数据中心的定时任务系统的。

第25章描述了批量数据处理系统的几种形态：从定时运行的MapReduce程序，到实时数据处理系统。不同的系统架构可带来不同的新奇与反直觉的挑战。

在第26章中，我们详细描述了数据一致性的重要性，如何确保之前存储的数据可以被安全地读取回来。

产品设计

最后，在第27章中，我们描述了处于整个可靠度金字塔模型顶端的产品设计理念。我们解释了Google是如何改进产品设计，以确保全球用户在产品上线第一天起就拥有最好的用户体验的。

扩展阅读

如前文所述，有效的测试（这里的测试同时指软件层面的测试，也指流程、组织上的灾难演习）是非常困难的。如果测试的方法不对，甚至会对整个服务的可靠性带来影响。在一篇ACM 论文中（参见文献[Kri12]），我们解释了Google是如何通过全公司的灾难演习来确保公司可以在大型灾难发生时正常工作的。

容量规划，有时候经常被说成是一种通过大量神奇的表格进行的黑魔法。但是通过文献[Hix15a]中的这篇文章，我们详细描述了正确地进

行容量规划并不一定需要用水晶球预见未来的能力。

最终，文献[War14]描述了Google的一种新的网络安全方法论。通过这项计划，我们正在逐步将有高权限的内网验证替换为使用用户设备和用户身份验证。当读者进行下一个网络设计的时候，可以试验一下这种方式。

第10章 基于时间序列数据进行有效报警

作者：**Jamie Wilkinson**

编辑：**Kavita Guliani**

让查询来得更猛烈些吧，让寻呼机[2]永远保持沉默！

——SRE 谚语

监控，处于整个生产环境需求金字塔模型的最底层。监控是运营一个可靠的稳定服务不可缺少的部分。服务运维人员依靠监控数据对服务的情况做出理性判断，用科学的方法应对紧急情况。同时，监控数据也可以用来确保服务质量与产品目标保持一致（参见第6章）。

不管一个服务目前是否由SRE运维，该服务都应该建立起对应的监控系统。SRE由于长期负责保障Google生产环境的运维，对支撑监控系统的基础设施非常了解。

监控一个大型系统本身是一项非常具有挑战性的工作：

- 大型系统中组件数量特别多，分析工作繁杂繁重。
- 监控系统本身的维护要求必须非常低。

Google的监控系统不仅要分析一些简单的系统指标（比如一台在欧洲Web服务器的平均响应时间），还需要分析更高抽象级别的指标（例如整个欧洲地区Web服务器的响应时间分布情况）。这类更高级的抽象

指标可以帮助我们寻找导致延迟长尾效应的问题所在。

在我们这种系统部署规模下，任何一个单机问题的报警都没有任何意义，因为这样的报警发生的次数太频繁，没有任何可操作性。我们采用的是另外一套方法：设计我们运维的系统使其能够更好地应对所依赖系统的故障。一个大型系统不应该要求运维人员持续关注其中使用的无数个小组件，而是应该自动汇总所有的信息，自动抛弃其中的异常情况。监控系统应该主要从高级服务质量目标层面进行报警，但是也应该保持足够的粒度，可以追踪到某个具体组件。

Google的监控系统经过10年的发展，从传统的探针模型（使用脚本测试，检查回复并且报警）与图形化趋势展示的模型已经演变为一个新模型。这个新模型将收集时间序列信息作为监控系统的首要任务，同时发展了一种丰富的时间序列信息操作语言，通过使用该语言将数据转化为图表和报警取代了以前的探针脚本。

Borgmon的起源

在Google的任务管理系统Borg（参见文献[ver15]）2003年面世之后不久，工程师就创造了一个新的监控系统，Borgmon。

Google之外的时间序列监控系统

这一章描述了Google内部使用了10年的监控工具的架构和编程接口。但是读者们一定想知道在Google外部如何使用它呢？

近年来，监控系统也经历了一次类似神武纪生物系统大爆炸的爆发式增长：Riemann、Heka、Bosun、Prometheus 都是开源软件中与Borgmon 基于时间序列数据报警理念类似的系统。Prometheus[3]与Borgmon十分类似，尤其是当你对比其中的规则计算语法时。变量收集和规则计算的理念在所有这些项目中都很类似。希望借助这些工具，读者可以自行试验、部署本章内描述的一些想法。

Borgmon 不使用特定的脚本来判断系统是否正常工作，而是依靠一种标准数据分析模型进行报警。这就使得批量、大规模、低成本的数据收集变得可能，而且不需要执行复杂的子进程以及建立特殊的网络连

接。我们将这种监控称之为白盒监控（参见第6章，有关“白盒监控”和“黑盒监控”的对比）。

收集回来的数据同时用来渲染图表和报警。报警规则用简单的数学表达式形式表达。因为数据收集过程不再是一个短暂的一次性过程，所有收集回来的数据历史都可以被用来作为报警规则的计算元素。

这些功能满足了第6章提到的简单性要求。它们使得整个监控系统的运行维护成本很低，从而可以保障运维服务人员有精力和资源应对服务规模的扩大以及部署变更带来的监控系统的调整。

为了更好地进行批量收集工作，我们为监控指标制定了一个标准格式。利用对varz[4]的一次HTTP请求，我们就可以收集到一个监控对象的所有监控指标。例如，如果想要查看webserver的所有监控指标，可以使用如下命令：

```
% curl http://webserver:80/varz
http_requests 37
errors_total 12
```

Borgmon 也可以从其他Borgmon实例上收集信息，[5]所以我们可以建立起一套和服务部署模型类似的层级体系。这样我们就可以逐级汇总监控指标，同时在每一级抛弃无用的信息。一般来说，运维团队在每个集群中会运行一个Borgmon实例，在全球范围（global）内再运行两个对等的Borgmon实例。[6]

一些部署规模非常大的服务甚至在集群内部部署了一些专门收集信息的Borgmon实例（scraper），这些实例用于收集信息，而集群实例则用于汇总。

应用软件的监控埋点

/varz这个HTTP接口只是用文本方式每行一个地列出应用中所暴露的全部监控变量值，格式是空格分隔的键值对。随后，又增加了一种Map格式，允许应用程序在键值对上增加标签（label）。例如下面这个例子，展示了25个HTTP 200响应，以及12个 HTTP 500响应。

http_responses map:code 200:25 404:0 500:12

增加一个监控指标只需要在具体服务程序中增加一行代码声明。

不难看出，采用这种无格式的文本形式作为监控的接口使得增加新监控点的门槛变得特别低，这对SRE和软件研发部门来说都是好事。但是，这同时也是对可维护性做出的一种妥协。监控指标的定义与Borgmon中对监控指标的使用分离意味着需要有人小心地维护相关改动（很容易发生的情况就是应用程序删除了某些指标，或者改变了具体含义，需要人工重新修改在Borgmon规则中的定义）。在实际使用中，我们开发了一个工具，用来自动校验Borgmon 规则的正确性，[7]从而很大程度上避免了这个问题。

如何将内部变量暴露给监控系统

Google内部产生的每个二进制文件中都默认包含一个HTTP服务，[8]同时每种主流编程语言都提供了一个编程接口可以自动注册暴露指定的程序变量。软件服务器作者可以随后对这个变量进行简单地增加操作，或者直接修改这个变量的值。Go语言中的expvar标准库[9]和它的JSON输出格式也提供了类似的API。

监控指标的收集

首先，Borgmon实例的配置文件中配置了需要收集的目标列表，目标位置可以使用各种地址解析服务支持的格式。[10]这个目标列表通常是动态变化的，所以一套服务发现体系可以降低监控系统的配置维护难度，允许监控系统自动扩展。

Borgmon 按照配置规定的周期，定时抓取/varz URI。将得到的信息解码，存储在内存中。Borgmon能够自动将每个目标的收集工作均匀地分散在整个周期内。在多级Borgmon的配置中，这样做可以避免上游Borgmon总是在同一个时间间隔抓取下游Borgmon的信息。

Borgmon同时还为每个监控目标记录了一些自动生成的“合成指标”，以便区分以下几种情况：

- 目标地址是否成功解析为IP和端口。
- 目标是否响应了一次收集请求。
- 目标是否响应了一次健康检查请求。
- 数据收集成功结束的时间点。

这些自动生成的监控指标，可以用在检测被监控任务不可用状态的规则编写中。

很有趣的一点是，varz 体系和SNMP（简单网络监控协议）非常不同。SNMP协议在设计中需要最简单的传输协议支持，保障在其他网络应用失败的时候也能正常工作（参见文献[Mic03]）。利用 HTTP协议收集监控信息好像和这个设计理念正好相反。但是，实践经验告诉我们这不是一个问题。[11]整套监控系统已经可以很好地应对网络和物理服务器故障，同时Borgmon还允许开发者使用上文提到的合成监控指标写出更好的报警规则。

时间序列数据的存储

一个服务通常由很多软件服务器组成，运行在分布于很多集群的物理服务器上。Borgmon需要将所有收集到的信息统一整理存储，同时允许灵活地查询相关数据。

Borgmon将所有数据保存在一个内存数据库中，定时保存到硬盘上。这些数据都是以类似（timestamp,value）的格式存储在一个按时间排序的链表里，该链表称为Ti m eseries。同时，每个t i m e-s e r i e s链表用一组唯一的标签命名（name＝value）。

如图10-1所示，一个time-series链表实际上是一个单维数字矩阵，以时间为Y轴。当我们给time-series加上各种标签时，这个矩阵就变成多维矩阵了。

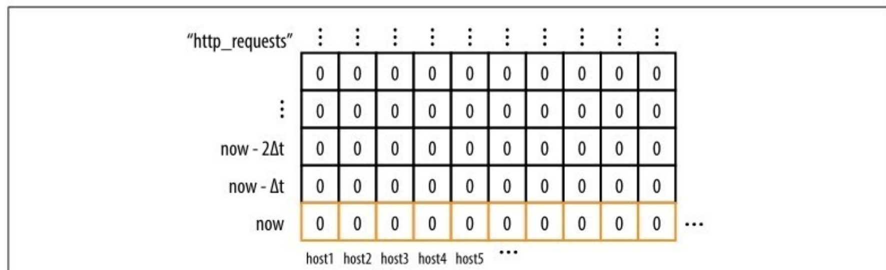


图10-1：从不同服务器上收集到的错误监控变量的存储形式

在实际实现中，这个数据接口存放在一个固定大小的内存块中，我们称之为time-series存放区（arena）。time-series 存放区存满后，同时有一个垃圾回收器，将过期的数据从内存中清除。time-series中最老和最新的数据间隔称之为数据地平线距离（horizon）。数据地平线距离代表了在内存中存放了多久的历史数据可供查询。通常情况下，数据中心和全局Borgmon中一般至少需要存放12小时左右的数据量，以便渲染图表使用。[12]下游的数据收集Borgmon，可以存放得更少。每一个数据点大概占用24字节的内存，所以存放100万个time-series，每个time-series 每分钟一个数据点，同时保存12小时的数据，仅需17GB内存。

Borgmon 会定时将内存状态归档至一个外部数据库（TSDB）。Borgmon可以通过查询TSDB获取历史数据。虽然TSDB容量要比Borgmon内存更便宜，也更大，但是查询速度会更慢。

标签与向量

正如图10-2的示范数据所示，time-series是按照时间戳和值的序列存放的，我们称之为向量（vector）。就像线性代数中的向量一样，这些向量是一个存放在time-series存放区中的多维矩阵中的某一列，或者是某一个对角线数值串。理论上来说，我们可以在描述这个数据结构时，忽略时间戳。因为序列中的每个值都是按照固定间隔插入的，不管是1s，还是10s，相邻的值的时间差是固定的。

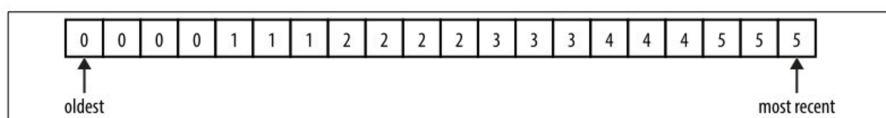


图10-2:time-series示范数据

time-series的名字称为标签集合（labelset），因为它的实现方式就是一个标签（key＝value）的集合。其中一个标签是变量名称，也就是varz页面中显示的键名。

在实现中，有一些标签被标记为重要的（important）标签（这里更多的是指代码中的一个分类，重要的标签会建有特殊索引，可加快查找速度）。在time-series数据库中，标识一个time-series的标签必须同时有以下几个标签：

var

代表变量名称。

job

被监控的软件服务器类型名。

service

一个松散定义的软件服务器类型组名，可以按对外名称分类，也可以按对内名称分类。

zone

Google定义的一个惯例名称，代表收集该条信息的Borgmon所在的位置（一般以数据中心名称赋值）。

这4条综合起来形成了一个完整的变量表达式（variable expression）：

{var＝http_requests,job＝webserver,instance＝host0:80,service＝web,zo
west}

当对time-series进行查询时，不一定要指定所有的标签。如果查询中指定了一个标签集合，那么所有符合这个标签集合的time-series都会被查询出来，形成一个向量返回。举例说明，

如果在一个集群中，有超过一个任务实例，那么下面的查询会返回

多个结果：

{var=http_requests,job=webserver,service=web,zone=us-west}

查询结果将会是一个向量，其中包含了每个符合条件的time-series的最后一个数值：

```
{var=http_requests,job=webserver,instance=host0:80,service=web,zone=us-west} 10
{var=http_requests,job=webserver,instance=host1:80,service=web,zone=us-west} 9
{var=http_requests,job=webserver,instance=host2:80,service=web,zone=us-west} 11
{var=http_requests,job=webserver,instance=host3:80,service=web,zone=us-west} 0
{var=http_requests,job=webserver,instance=host4:80,service=web,zone=us-west} 10
```

标签的来源有：

- 监控目标的名称，如job 和 instance 来源于任务名和实例地址。
- 监控目标自行提供，如提供的Map类型变量。
- Borgmon配置文件，其中可以添加和替换标签。
- Borgmon 规则。

同时，通过在变量表达式后面增加一个时间参数，可以查询一段时间内的全部timeseries值：

{var=http_requests,job=webserver,service=web,zone=us-west}[10m]

该条查询返回过去10分钟内，所有满足条件的time-series。如果每分钟收集一次监控指标，那么应该每条记录获得10个数值点：[13]

```
{var=http_requests,job=webserver,instance=host0:80, ...} 0 1 2 3 4 5 6 7 8 9 10
{var=http_requests,job=webserver,instance=host1:80, ...} 0 1 2 3 4 4 5 6 7 8 9
{var=http_requests,job=webserver,instance=host2:80, ...} 0 1 2 3 5 6 7 8 9 9 11
{var=http_requests,job=webserver,instance=host3:80, ...} 0 0 0 0 0 0 0 0 0 0 0
{var=http_requests,job=webserver,instance=host4:80, ...} 0 1 2 3 4 5 6 7 8 9 10
```

Borg规则计算

Borgmon从本质上来说，是一个可编程计算器，并且加入了一些语法糖，从而可以让它产生报警信息。数据收集部分和存储部分都是为这

个可编程计算器而服务的。



将规则计算集中到一个监控系统内完成，而不是分散到多个子进程中，可以方便将同样的计算在很多不同的监控目标上同时应用。这种设计方式可确保配置文件相对较小（可以大幅去掉重复代码），同时也可确保足够强大的表达能力。

Borgmon编程语言，称为Borgmon规则，由简单的代数计算表达式组成。这些代数计算表达式使用一个time-series 作为输入，计算出另外一个time-series。这些规则的功能很强大，因为它们可以将计算应用在单个time-series的历史数据（时间维度）上，也可以同时应用在很多个不同的time-series（空间维度）上。

Borgmon 规则一般运行在一个线程池中，但是这受限于规则定义的输入是否包含前序规则的输出。每个查询表达式的结果向量大小决定了这条规则的执行速度。一般来说，Borgmon的运行速度可以通过增加新的CPU资源而提高。为了帮助深入分析性能问题，Borgmon还提供了一系列内部规则计算的详细监控指标。

汇总计算（aggregation），是分布式环境中不可缺少的一环。汇总计算过程可以将一个任务的所有实例中的某个time-series相加。通过计算总数，我们就可以计算整体速率（rate）。例如：一个任务在整个数据中心中的整体每秒查询率（QPS）需要通过所有实例的查询计数器[14]的变化率的总和[15]来计算。



一个计数器（counter）的值应该永远是上涨的，或者精确地说，它的值应该是非单向递减的。[16]相比之下，测量器（gauge）类型在不同时刻则可能有任意值。计数器通常用来衡量单向递增的变量，例如行驶公里数。而测量器用来显示当前状态，例如目前剩余油量，或者目前行驶速度等。当收集数据时，我们鼓励多采用计数器模式收集。因为计数器模型不会在两次采集间隔中间丢失信息。如果采用测量器模型，两次采集周期之间，可能会错过某些数值变化的情况。

举例来说，针对一个Web服务器，我们想在Web服务器返回错误回复超过一定比例的时候产生报警。具体来说，就是集群中所有实例的非HTTP 200回复的速率总和除以所有请求的速率总和超过某个数值的时候，发出报警。

有以下三个步骤：

1.汇总所有实例的HTTP回复的速率，按回复代码分类，计算出一个向量。

2.计算所有的“错误回复”的速率总和，得出一个单值，作为整个集群的错误速率。在求和计算中排除了状态值为200的速率，因为这是正确回复。

3.计算整个集群的错误速率比例，用错误回复的速率除以所有请求的速率总和，再次产生一个单值。

每一个步骤的输出都会被写入一个time-series变量中，变量名称是由这条规则的变量名表达式指定的。变量被记录后，我们可以通过名称定位这些结果。

下面是计算所有请求的速率总和的Borgmon规则：

```
rules <<<
  # 用每个任务实例的请求计数器请求速率。
  {var=task:http_requests:rate10m,job=webserver} =
    rate({var=http_requests,job=webserver}[10m]);

  # 对所有实例的速率求和，得出整体汇总速率。
  # without instance 指示 Borgmon 将 instance label 从表达式右侧统一排除。
  {var=dc:http_requests:rate10m,job=webserver} =
    sum without instance({var=task:http_requests:rate10m,job=webserver})
>>>
```

rate（ ）函数使用提供的表达式输出作为输入，返回总体差值除以总时间。

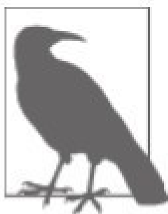
用之前的一个例子，我们可以看到task:http_requests:rate10m的结果是：[17]

```
{var=task:http_requests:rate10m,job=webserver,instance=host0:80, ...} 1
{var=task:http_requests:rate10m,job=webserver,instance=host2:80, ...} 0.9
{var=task:http_requests:rate10m,job=webserver,instance=host3:80, ...} 1.1
{var=task:http_requests:rate10m,job=webserver,instance=host4:80, ...} 0
{var=task:http_requests:rate10m,job=webserver,instance=host5:80, ...} 1
```

而dc:http_requests:rate10m的结果是：

```
{var=dc:http_requests:rate10m,job=webserver,service=web,zone=us-west} 4
```

因为第二条规则使用第一条规则的结果作为输入。



instance　　标签在结果中已经消失了，因为汇总过程中已经将其去除。如果没有去除，Borgmon无法将所有的行加到一起。[18]

在这个例子中，我们使用了一个时间窗口值（time　window），也就是上文中的10m，因为我们的操作对象是一个一个独立的值，而不是

一个连续函数。这样做的好处是，计算速率时简化了计算，不需要进行微积分运算。但是这样同时也要求我们保证有足够的数据点，才能计算出准确的速率。尤其要考虑的是，近期几次数据收集可能失败的情况。所以上文中查询表达式中我们使用了[10m]这种区间形式来避免某些数据缺失的情况。[19]

在前面的例子中，变量命名采用了Google惯例，有助于提高可读性。每一个计算得出的变量名必须包括一个由“:”分隔的组，分别显示汇总级别、变量名称，以及创建这个变量的操作。在这个例子中，表达式左侧变量名的意思是“实例级别 HTTP 请求的10分钟内速率”和“数据中心级别 HTTP 请求的10分钟内速率”。

现在我们知道如何计算请求速率了，可以按这个方式计算出错误回复的速率，同时计算出错误回复速率和请求速率的比率，从而得出Web服务器的健康程度。可以将这个值作为我们的服务质量目标（SLO，参见第4章）。同时针对这个服务质量目标进行报警，报警条件可以是已经越过SLO指标的情况，或者是即将越过SLO指标的情况。

```
rules <<<
  # 按 code 标签给每个实例计算速率
  {var=task:http_responses:rate10m,job=webserver} =
    rate by code({var=http_responses,job=webserver}[10m]);

  # 按 code 标签计算一个集群级别的汇总速率
  {var=dc:http_responses:rate10m,job=webserver} =
    sum without instance({var=task:http_responses:rate10m,job=webserver});

  # 计算非 200 代码的集群汇总速率
  {var=dc:http_errors:rate10m,job=webserver} = sum without code(
    {var=dc:http_responses:rate10m,job=webserver,code=!/200/};

  # 计算错误速率和请求速率的比例
  {var=dc:http_errors:ratio_rate10m,job=webserver} =
    {var=dc:http_errors:rate10m,job=webserver}
      /
    {var=dc:http_requests:rate10m,job=webserver};
>>>
```

同样的，上述计算展示了按照Google惯例命名的变量名。结果含义为“数据中心级别HTTP错误的10分钟内速率的比例”。

最终结果的输出可能是下面这样：[20]

```
{var=task:http_responses:rate10m,job=webserver}

  {var=task:http_responses:rate10m,job=webserver,code=200,instance=host0:80, ...} 1
  {var=task:http_responses:rate10m,job=webserver,code=500,instance=host0:80, ...} 0
  {var=task:http_responses:rate10m,job=webserver,code=200,instance=host1:80, ...} 0.5
  {var=task:http_responses:rate10m,job=webserver,code=500,instance=host1:80, ...} 0.4
  {var=task:http_responses:rate10m,job=webserver,code=200,instance=host2:80, ...} 1
  {var=task:http_responses:rate10m,job=webserver,code=500,instance=host2:80, ...} 0.1
  {var=task:http_responses:rate10m,job=webserver,code=200,instance=host3:80, ...} 0
  {var=task:http_responses:rate10m,job=webserver,code=500,instance=host3:80, ...} 0
  {var=task:http_responses:rate10m,job=webserver,code=200,instance=host4:80, ...} 0.9
  {var=task:http_responses:rate10m,job=webserver,code=500,instance=host4:80, ...} 0.1

{var=dc:http_responses:rate10m,job=webserver}

  {var=dc:http_responses:rate10m,job=webserver,code=200, ...} 3.4
  {var=dc:http_responses:rate10m,job=webserver,code=500, ...} 0.6

{var=dc:http_responses:rate10m,jobwebserver,code=!/200/}

  {var=dc:http_responses:rate10m,job=webserver,code=500, ...} 0.6

{var=dc:http_errors:rate10m,job=webserver}

  {var=dc:http_errors:rate10m,job=webserver, ...} 0.6

{var=dc:http_errors:ratio_rate10m,job=webserver}

  {var=dc:http_errors:ratio_rate10m,job=webserver} 0.15
```



这里的输出展现了中间结果dc:http_errors:rate10m这条规则过滤了非200状态值。虽然值并没有改变，但是可以观察到code标签从中消失了。

正如前文所述，Borgmon 规则最终创建了新的time-series，以便将计算结果保存起来供未来使用。在实践过程中，Borgmon支持临时查询，可以将结果以表格或者图表方式展现，在调试中这非常有用。如果这些临时查询被证实有用，它们可以被存储下来，写入Borgmon 规则配置文件中，作为服务控制台（console）上的一个永久图表。

报警

每当Borgmon计算完成一条报警规则时，结果永远是真（true）或假（false）。如果结果为真，那么产生一条报警。经验证明，报警规则经常反复变动（flap，快速切换状态）。因此，每条报警规则都指定了一个最小持续时间值。只有当警报持续时间超过这个值的时候，才会发送警报。一般来说，这个周期至少被设置为两个计算周期，以确保偶尔一次的信息收集失败不会立刻触发报警。

下面这条报警规是在10分钟内错误速率比率超过1%的时候，同时整体错误速率超过1（这里的单位为1/s,rate函数的结果都是以秒为单位）的时候触发一条报警信息：

```
rules <<<
  {var=dc:http_errors:ratio_rate10m,job=webserver} > 0.01
    and by job, error
  {var=dc:http_errors:rate10m,job=webserver} > 1
    for 2m
    => ErrorRatioTooHigh
      details 'webserver error ratio at [[trigger_value]]'
      labels {severity=page};
>>>
```

我们之前的例子的计算结果为0.15，已经超过了这里的阈值0.01。但是由于整体错误速率并没有超过　　1/s，所以并没有触发这条报警规则。一旦整体错误速率超过1，该条报警会进入等待（pending）状态，确保不会反复变动，直到超过2分钟之后，才会触发报警。

报警规则本身包含了一个小模板，以供产生具体报警错误信息时使用。里面包含有任务名称、报警名称，以及触发报警时实际的值（trigger_value）等。这些上下文信息由Borgmon发送Alert　RPC的时候提供。

Borgmon连接到一个全局共享的服务，Alertmanager（报警管理服务）。报警管理服务负责接收Alert RPC。报警进入等待状态和触发状态（firing）时都会产生Alert　RPC通知报警管理服务。报警管理服务负责将收到的报警转发到合适的通知渠道。报警管理服务的配置包括：

- 当有其他报警触发的时候，抑制某些报警。

- 将多个Borgmon发来的报警信息合并排重。
- 根据标签信息将收到的报警信息展开或者将多个报警信息合并成一个。

如第6章所述，每个SRE团队都会将严重情况报警发送给当前on-call工程师。将重要但不紧急的报警发送给工单系统。其他报警一般用来作为历史数据或者服务监控台（dashboard）展示使用。更详细的报警策略设计规则请参见第4章。

监控系统的分片机制

一个Borgmon实例可以从另外一个实例中获取信息。虽然用一个实例收集全世界范围内运行的某个服务的全部实例在理论上可行，但在实际操作中很快就会遇到性能问题。同时，这样也会成为设计中的单点故障源。为此，我们设计了一种流式传输协议，用于Borgmon之间互相传输time-series数据，进而节约不断收集服务数据带来的CPU和网络成本。一个标准部署模型中包括两个甚至更多的全局Borgmon，负责进行全局汇总。每个数据中心中运行一个Borgmon，负责汇总所有在该数据中心中运行的任务实例。（Google将生产环境按区域划分，任何改动都以区域为单位进行。运行两个实例可以保障在数据中心维护或者意外情况下产生的单点故障问题。）

如图 10-3所示，在更复杂的部署模型中，我们进一步将数据中心Borgmon划分为一个收集层和一个汇总层（一般由于单个Borgmon的RAM和CPU限制导致）。数据收集层主要负责规则运算和汇总。有的时候全局层（global）也会被划分为计算层和展示层（展示层需要处理动态查询，十分消耗内存）。上游Borgmon 可以过滤从下游Borgmon收集来的信息，这样全局Borgmon就不需要保留所有任务实例层面的time-series信息。这样一来，一个按层级汇总的体系就建立起来了，也允许根据需要逐级下行去取得底层的信息。

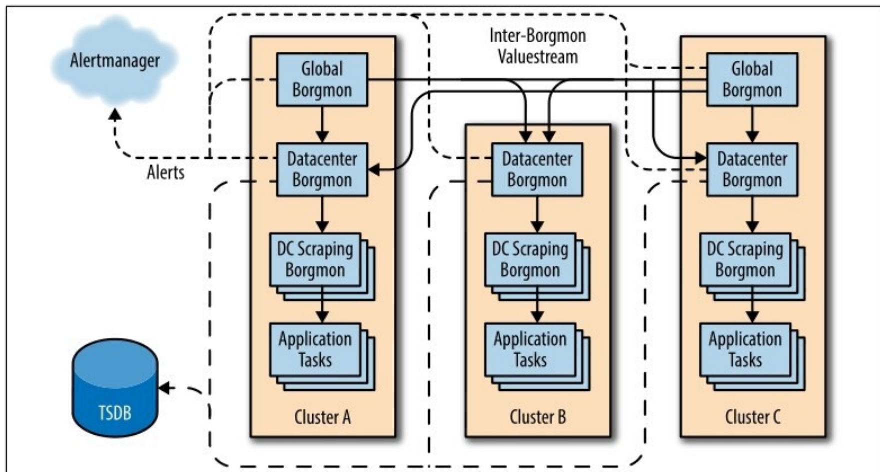


图10-3：一个在三个集群中层级部署的Borgmon 数据流向图

黑盒监控

前文说过，Borgmon是一个白盒监控系统，负责监控目标服务的内部状态。同时，Borgmon规则也是基于服务内部指标写成的。这种透明模型可以非常有效地找到错误根源，无论是对应的组件失败，某个队列满了，还是系统中存在某种性能瓶颈。无论在新功能部署时还是应对紧急情况下都非常有用。

但是，白盒监控并不能完全代表一个被监控系统的所有状态。完全依赖白盒监控，就意味着我们并不知道最终用户看到的是什么样。例如：白盒监控只能看到已经接收到的请求，并不能看到由于DNS故障导致没有发送成功的请求，或者是由于软件服务器崩溃而没有返回的错误。同时，报警策略也只包含了工程师能想到的错误情况。

Google SRE团队通常利用探针程序（prober）解决了该问题。探针程序使用应用级别的自动请求探测目标是否成功返回。这些探针可以直接向报警管理服务发送Alert RPC，也可以由Borgmon收集其结果信息。探针程序还可以针对返回结果进行应用级别的校验。例如检查HTTP回复中的HTML结果。探针程序甚至可以将结果中的值取出作为timeseries暴露给Borgmon使用。SRE团队经常使用探针程序收集响应速度的直方图和回复大小，以便观察最终用户可见的性能指标。探针程序是一种检查测试模型和一些高级time-series创建功能的混合体。

探针程序既可以直接探测前端，也可以探测负载均衡服务后面的服务。通过对两种不同情况的探测，我们可以发现局部问题并且消除无效报警。举例来说，我们可能同时检测负载均衡器前面的www.google.com以及负载均衡器后面的真实We b服务器。在这种模式下，我们可以知道当某个数据中心出现问题时，整个www.google.com依然工作，同时也可以在发生全局问题时，快速定位某个具体的数据中心。

配置文件的维护

Borgmon配置文件是将规则定义与具体收集的目标分开组织的。这意味着同样一套规则可以应用到许多不同的收集目标上。这种分离机制可能看起来没有那么重要，但是却极大降低了监控系统配置文件的维护要求。

Borgmon同时支持语言级别的模板。Borgmon提供一种类似于宏的模板机制，允许用户创造出一些可重用的规则库，进一步减小了配置文件中的重复程度，降低配置文件出错的可能性。

当然，任何高级编程语言环境中都难以避免复杂程序出现Bug,Borgmon为此提供了针对Borgmon规则的单元测试和回归测试工具。通过使用这些工具，以及一些合成的time-series数据，我们可以确保每条Borgmon规则都是正确的。生产环境监控架构SRE组（production monitoring team）维护一套持续集成体系，针对所有配置文件运行一系列测试，然后统一分发到所有的Borgmon实例上。Borgmon 实例会在接受新配置之前进行校验。

在Google内部大量的Borgmon通用模板库中，两类模板是最多的。第一类模板将某一个代码类库暴露的所有监控信息系统模板化，从而使得使用这个类库的用户可以通过使用这个模板构建监控系统。这样的模板包括 HTTP 服务器类库、内存分配器类库、存储系统客户端类库及通用RPC框架等。（虽然varz 接口并没有要求任何命名体系，但是Borgmon规则类库和代码类库形成了一套事实意义上的命名体系。）

第二类模板代表了一种如何将单实例监控数据按级汇总到全局范围

的模型。这些模板提供了一系列通用汇总模板，允许用户将任意指定监控值按自己的服务部署模型汇总。

例如，一个服务可能提供了一个全球可用的API，由处于多个数据中心的任务实例组成。在每个数据中心中，服务由多个分片（shard）组成，每个分片由多个任务组成，同时每个任务有不同的实例数量。一个工程师可以将这个层级关系体现在Borgmon体系中，使得调试单独组件时可以单独获取对应信息。这种分组结构一般是根据命运共享（share of fate）理念进行的。例如一个任务的所有实例是命运共享的，由于它们共享一个配置文件（可能会由于错误的配置文件而同时失败）。而一个分片中的任务也是命运共享的，因为它们都运行在同一个数据中心中，而数据中心由于使用同一个网络结构而有可能同时失败。

通过使用Borgmon标签机制，我们可以针对任务进行区分：Borgmon给每个目标添加对应的实例名称、分片，以及所在的数据中心。这些标签可以用来分组和汇总对应的timeseries。

因此，标签在整个系统中有很多用途：

- 标签可以用来标记数据维度（例如，HTTP 状态码信息用code标签在http_responses变量中标记）。
- 标签可以标记数据来源（实例名称和任务名）。
- 标签可以标记局部分组信息和汇总信息（例如zone标签代表数据中心，shard标签代表逻辑分组信息）。

这些模板库非常灵活，一套模板可以在不同的汇总级别上应用。

十年之后

Borgmon将测试和报警模型革命成了海量信息收集和中央化规则计算、统一分析和报警的新模型。

这种松耦合模式使得被监控系统可以独立于监控系统报警规则之外自由扩展。同时由于这些报警规则都基于一种抽象的time-series模型，

也更易于维护。新应用程序上线之时就已经带有了使用的类库中的监控信息，同时大量使用模板化的汇总规则和监控台模板，也让具体监控变得很简单。

保证监控系统的维护成本与服务的部署规模呈非线性相关增长是非常关键的。这是在SRE工作中不断重现的一个模式，SRE确保他们做的每一件事都能够扩展到更大的规模。

十年是一个相当长的时间了，随着Google的不断成长，今天Google内部的监控系统也在持续不断地改进和完善。

虽然Borgmon仍是Google内部工具，但是任何人都可以利用开源软件尝试这种使用time-series进行监控和报警的理念。Pormetheus、Riemann、Heka和Bosun都是很好的例子。

第11章 on-call轮值

作者：**Andrea Spadaccini** [21]

编辑：**Kavita Guliani**

on-call轮值是很多运维和研发团队的重要职责，这项任务的目标是保障服务的可靠性和可用性。但是，on-call轮值制度执行过程中有一些十分重要的环节，如果没有正确执行，将会给服务甚至团队带来非常严重的后果。本章描述了Google SRE历年来执行on-call轮值制度时发展的核心方法论，解释了这些方法论是如何使服务变得更可靠，工作压力变得更均衡的。

介绍

很多职业要求从事该职业的人员参与某种类型的on-call轮值任务。在on-call值班过程中，值班人员必须保证可以随时响应紧急问题，不管是在工作时间，还是在非工作时间。在IT行业里，on-call任务一般是由专门的运维团队（Ops）进行的，主要目标是保障他们所运维的服务维

持正常运转。

Google内部的很多重要服务，比如Search、Ads、Gmail等都有专门的SRE团队负责性能和可靠性保障。因此SRE　要参与所服务项目的on-call轮值工作。SRE团队和纯运维团队十分不一样的地方在于，SRE团队非常强调用工程化手段来应对运维问题。而这些运维问题，当达到一定规模时，也确实只有采用软件工程化手段才能解决。

为了鼓励这种工程化解决问题的方式，Google在招聘SRE团队时着重选拔同时具有系统工程经验和软件开发经验的人。我们为SRE花在纯运维事务上的时间设立了50%的上限。SRE至少要花50%的时间进行工程项目研发，以便能够研发出更好的自动化和服务优化手段来更好地服务整个业务。

on-call工程师的一天

这一节主要介绍了on-call工程师的主要工作，为本章其余部分做背景介绍。

作为生产系统的监管者，on-call工程师负责处理生产环境中即将或者正在发生的业务事故，以及评审对生产系统的变更请求。

在on-call时，该工程师承诺可以在分钟级别执行生产系统中的维护需求，具体的时间范围是SRE团队和产品业务团队共同商议得出的。一般来说，对任何直接面向最终用户的服务，或者时间非常紧迫的服务来说，这个时间定为5分钟。而非敏感业务通常来说是30分钟。公司提供一个可以接受报警信息的设备，通常来说是手机。Google有一套非常灵活的报警传递机制，可以将报警信息通过各种不同渠道同时送达多个设备（邮件、短信、自动电话呼叫及APP等）。

响应时间通常跟业务需要的可靠性有关。比如在下面这个简单例子中：如果一个面向最终用户的业务系统在每个季度想要达到99.99%的可用度，那么每个季度共有13分钟的不可用时间（参见附录A）。这个要求说明了on-call工程师必须在分钟级别上响应生产事故（更确切地说，13分钟内）。如果一个系统的SLO更为宽松，那么响应时间可以在几十

分钟内。

一旦接收到报警信息，工程师必须确认（ack）,on-call工程师必须能够及时定位问题，并且尝试解决问题。必要的话，on-call工程师可以联系其他团队，或者升级（escalate）请求支援。

非紧急的生产系统事件，例如低优先级警报的处理，或者新软件的发布可以由on-call工程师在工作时间内评审或者执行。这些任务与生产报警信息相比不算紧急。生产报警信息的处理是第一紧急要务，几乎超过一切其他活动，包括研发项目的进行。如果想要了解更多有关运维事务压力的讨论，请参见第29章。

很多团队同时有主on-call人和副on-call人值班，主副分工每个团队都各有不同。有的团队可能以副on-call人作为主on-call人的辅助，负责处理主on-call人没有响应的情况。在另一个团队里，主on-call人只负责处理生产系统紧急情况，而副on-call人负责处理其他非紧急的生产环境变更需求。

在团队中，副on-call轮值制度并不是必需的。很多相关性密切的团队经常彼此作为副on-call，互相值班，共同分担工作压力。

on-call轮值制度的具体制定有很多种方式，请参看文献[Lim14]中关于on-call章节的更详细介绍。

on-call工作平衡

SRE团队对on-call工作的质量（这里更多是指工作压力）和数量有明确的要求。数量是指某个工程师在on-call事务上花费的具体时间。质量则可以通过每次on-call轮值发生的事故数量决定。

SRE管理者负责保证on-call轮值的工作压力在这两个维度保持在一个可持续的水平上。

数量上保持平衡

我们十分强调SRE中的“E”（Engineering），认为这是区别我们与

传统Ops团队的不同之处。所以，我们强调至少将SRE团队50%的时间花在软件工程上。在其余时间中，不超过25%的时间用来on-call，另外25%的时间用来处理其他运维工作。

利用25% 这个时间上限，我们可以计算出满足7×24 on-call轮值制度的最少SRE数量。假设每次on-call轮值中需要有两名工程师（主on-call和副on-call，分别进行不同的工作），则这个团队需要至少8名工程师。假设每次on-call值班长度为一周，那么每名工程师只需要每月轮值一次。如果团队分散在两地，那么最少工程师的数量是6名。这样既可以保障25%的时间上限，也可以保证两地各有足够的工程师开展研发工作。

如果一个服务有足够的业务需求需要更多的人，与其继续扩展当地团队（single site），我们更愿意把该团队转化为多地团队（multi site）。多地团队相比之下有以下两点优势：

- 长时间执行夜间任务对人的健康不利（参见文献[Dur05]）。多地团队可以利用“日出而作，日落而息”（Follow the sun）的轮值制度使整个团队避免夜间值班。
- 通过限制一个团队在on-call轮值制度中的人员数量，可保障每个工程师对生产环境的熟悉程度（请参见本章后面的“奸诈的敌人—运维压力不够”一节）。

但是，多地团队增加了沟通和合作的成本。因此，是否将当地团队转为多地团队应该慎重地决定，仔细考虑两种模式的成本，业务系统的重要性，以及每个系统的业务压力。

质量上保持平衡

每次on-call值班过程中，轮值工程师必须有足够的时间处理紧急事件和后续跟进工作，例如写事后报告（参见文献[Loo10]）。我们对一个紧急事件（incident）的定义是，一系列根本原因一致或者相关的事件和报警信息，这些事件应该在同一个事后报告中讨论。我们发现，平均下来，处理任何一个生产环境事件，包括事件根源分析、事件处理，

以及事后总结等，至少需要6个小时。因此，我们认为，每12个小时的轮值周期最多只能产生两个紧急事件。为了维持在这个范围内，报警事件的每日发生概率应该很低，中值保持在0左右。如果某个组件几乎每次轮值都会报警，导致每日紧急事件中值＞1，那么早晚另外一个组件也会同时报警，导致当天总报警次数超标。

如果在某个季度中，某个团队持续不断地超过这个指标，那么管理团队必须采取一些修正措施保证运维压力下降到可持续水平，具体可参见本章后面的“运维压力过大”一节，以及第30章）。

补贴措施

管理层需要考虑，针对工作时间之外的on-call工作应给予合理的补贴。不同的组织团队用不同的方式进行补贴。Google提供年假或者现金补贴，同时按一定程度的工资比例作为上限。这个限制实际上是为限制每个工程师的on-call时间比例。整套激励体制的建立是为了保障团队按需参与on-call，同时避免过量on-call带来的问题，例如项目开发时间不够，或者疲劳过度（burnout）。

安全感

正如前文所述，SRE负责运维Google最重要的生产系统。在on-call轮值期间，SRE工程师要直接负责用户可见、直接影响公司收入的系统，或者是支撑这些系统运转的基础设施系统。SRE思考和解决问题的方法论对正确处理问题是非常关键的。

现代理论研究指出，在面临挑战时，一个人会主动或非主动（潜意识）地选择下列两种处理方法之一（参见文献[Kah11]）：

- 依赖直觉，自动化、快速行动。
- 理性、专注、有意识地进行认知类活动。

当处理复杂系统问题时，第二种行事方式是更好的，可能会产生更好的处理结果，以及计划更周全的执行过程。

为了确保on-call工程师可以保持在第二种处理方式范围内，我们必须要减轻on-call所带来的压力感。业务系统的重要性和操作所带来的影响程度会对on-call工程师造成巨大的精神压力，危害工程师的身体健康，并且可能导致SRE在处理问题过程中犯错误，从而影响到整个系统的可靠性。从医学上讲，压力状态下释放的荷尔蒙，例如皮质醇和促肾上腺皮质素（CRH）都会对人的行为造成影响，甚至造成恐惧，进而影响人类进行正常认知功能的工作，最后导致错误行为的产生（参见文献[Chr09]）。

在这些压力释放的荷尔蒙的影响下，on-call工程师往往会选择反应性的、未经详细考虑过的操作，这些操作很容易导致“过度联想”现象的产生。“过度联想”是在on-call中非常容易产生的现象，举例来说，当on-call工程师收到本周内第4个同样报警信息时，很容易联想起前3次报警都是由于某个外部系统造成的虚假报警，于是很自然地将第4次报警也归类为虚假报警，从而没有认真处理，导致真实事故的发生。

在应急事故处理过程中，凭直觉操作和快速反应（例如服务出现问题就先重启服务器）看起来都是很有用的方法，但是这些方法都有自己的缺点。直觉很可能是错误的，而且直觉一般都不是基于明确的数据支持的。因此，在处理问题的过程中，on-call工程师很有可能由于凭直觉去解释问题产生的原因而浪费宝贵的时间。快速反应主要是由习惯而产生的，习惯性的快速反应的动作后果一般都没有经过详细考虑，这可能会将灾难扩大。在应急事件处理过程中，最理想的方法论是这样的：在有足够数据支撑的时候按步骤解决问题，同时不停地审视和验证目前所有的假设。

让on-call SRE知道他们可以寻求外部帮助，对减轻on-call压力也很有帮助。最重要的资源有：

- 清晰的问题升级路线。
- 清晰定义的应急事件处理步骤。
- 无指责，对事不对人的文化氛围（参见文献[Loo10]和[All12]）。

通常，SRE运维的生产系统对应的开发团队也会参与7×24 on-call轮值，所以SRE团队永远可以寻求这些伙伴团队的帮助。当非常严重、原因不明确的紧急情况出现时及时寻求开发团队的帮助常常是解决问题的关键。

一个工程师在处理非常复杂、需要同时引入多个团队的问题时，或者经过一段时间调查仍不能预测多久能够恢复时，应该考虑启用某种正式的应急事务处理流程。Google SRE使用的是在第14章中描述的流程。这个流程定义了一系列清晰的简单步骤，沿用这些步骤可以帮助on-call工程师利用所有可用的资源将问题最终解决。对应这套流程，Google内部开发使用了一个Web工具，可自动化大部分操作。比如提供方便的职责交接流程，以及保留对外状态更新公告的历史等。利用这些工具，on-call工程师可以专注于解决问题，而不是在格式化公告邮件，或者是同时更新多个对外沟通渠道上浪费时间。

最后，在应急事件处理结束时，仔细评估哪些地方有问题，哪些地方做得好是非常关键的。而且应该采取措施避免再次发生同样的问题。SRE团队必须在大型应急事件发生之后书写事后报告，详细记录所有事件发生的时间线。这些事后报告都是对事不对人的，为日后系统性地分析问题产生的原因提供了宝贵数据。犯错误是不可避免的，软件系统应该提供足够的自动化工具和检查来减少人为犯错误的可能性（参见文献[Loo10]）。

避免运维压力过大

前面，我们提到了SRE最多只花50%的时间在运维工作上。那么当运维压力超过这个限制的时候怎么办呢？

运维压力过大

当这种情况发生时，SRE团队和管理层负责在季度工作计划中加入一些应对措施，以保障工作压力可以恢复到可持续水平。从其他团队临时抽调一名有经验的SRE，常常可以帮助处于压力下的团队及时恢复常态（参见第30章）。

理想情况下，运维压力过载应该是基于数据且可以量化的。这样我们的目标也就可以量化（例如：每天工单数量□5，每次轮值报警事件 □2等）。

错误的监控系统配置常常是导致运维压力过大的原因。报警策略必须跟服务的SLO目标一致，每条报警信息必须是可实际操作的（指收到报警后有明确的动作需要执行）。一个每小时都触发的低优先级报警会严重影响on-call工程师的生产力。同时“狼来了”效应会导致真正重要的报警被忽略，具体可参看第29章。

同时，控制on-call工程师收到的针对同一起事故的报警总数也很重要。有的时候，一个异常情况可能会触发多条报警，所以合理地分组汇总报警信息是很重要的。通常，工程师在处理紧急事件时，会暂时禁止重复和无关报警，以便更好地专注在真正重要的工作上。如果某条报警规则经常产生重复报警，那么需要修改报警规则，将其调整为事件/报警比例接近1:1。这样，在问题真正发生时，工程师可以专注于解决问题，而不是重复地处理报警。

有的时候，造成运维压力上升的因素并不是由SRE控制的，比如研发团队可能修改了程序导致程序稳定性下降，或者更容易产生报警。在这种情况下，SRE需要和研发团队设立一个共同目标，解决运维压力问题。

在极端情况下，SRE团队可以选择停止支持某个服务。该服务将由开发团队负责on-call轮值，直到系统达到SRE团队设立的稳定性目标为止。但是这种情况发生的概率极小，因为基本上SRE总是可以和研发团队共同努力，使得运维压力下降。但是在某些情况下，降低系统运维压力可能需要改变复杂的系统架构，甚至需要几个季度的时间来完成。在这种情况下，SRE团队不应该承担全部运维压力，而是应该和研发团队协商，将某些或者全部报警信息转交给研发组处理。采用这样的临时措施，SRE才能有时间和精力与开发团队共同将系统运维压力降低，以便重新由SRE独立运维。

SRE和产品研发团队的协商机制，有助于平衡两个团队的话语权。[22]这种合作关系恰恰说明了一个良好的矛盾处理机制（系统可靠性与

功能上线速度）可以提升整个产品的质量，进而为公司创造更高的价值。

奸诈的敌人—运维压力不够

虽然给一个非常安静的系统on-call值班是很幸福的事情，但是当一个系统太稳定，或者SRE　on-call的周期太长会发生什么呢？SRE团队运维压力不够也是一个不良现象。长时间不操作生产环境会导致自信心问题，包括自信心太强以及自信心不够。这些现象只有在下一次发生问题时，才会显现出来。

为了避免这种问题，应该控制SRE团队的大小，保证每个工程师每个季度至少参与oncall一次，最好两次。这样可以保证团队成员有足够的生产环境操作经验。“命运之轮”（见第28章）也是一种有助提高技能和共享知识的团队活动。同时，Google每年举办一次持续数天的全公司灾难恢复演习（DiRT），针对理论性和实际性的灾难进行演练（参见文献[Kir12]）。

小结

本章中讨论的on-call轮值制度，是Google　SRE所有团队的指导思想，这个制度保障了一个可持续的运维工作环境。这套模型使Google可以用工程化的手段解决服务扩展性问题，在服务变得越来越复杂、SRE团队负责的组件越来越多的情况下仍可保持相当高的可用性和可靠性。

虽然本文提到的理念不一定可以立即应用在IT行业的所有需要on-call的领域内，但是我们相信，这套理念为面对行业内日益增长的on-call需求提供了一个可靠的参考模型。

第12章 有效的故障排查手段

作者：**Chris Jones**

值得警惕的是，理解一个系统应该如何工作并不能使人成为专家。

只能靠调查系统为何不能正常工作才行。

——Brian Redman

系统正常，只是该系统无数异常情况下的一种特例。

——John Allspaw

故障排查是运维分布式计算系统的一项关键技能。这项技能通常被认为是与生俱来的，有些人有，有些人没有。造成这种误解的原因通常是，因为对一个经常需要进行排除故障的人来说，这是一个根深蒂固的理念，让他去解释“如何”去进行故障排查是一件很难的事情。这就像解释如何骑自行车一样。但是在本章中，我们认为故障排查过程是一个可以自我学习，也是一项可以传授的技能。

新手们常常不能有效地进行故障排查，是因为这个过程理想情况下同时需要两个条件。1.对通用的故障排查过程的理解（不依靠任何特定系统）。2.对发生故障的系统的足够了解。虽然只依靠通用性的流程和手段也可以处理一些系统中的问题，[23]但我们发现这样做通常是很低效的。对系统内部运行的了解往往限制了SRE处理系统问题的有效性，对系统设计方式和构建原理的知识是不可或缺的。

让我们一起来看一下基本的故障排查流程。对这个流程非常熟悉的读者可以不用太在意我们的流程定义和名词使用。如果你现在的方法很好用，没有任何理由去改变。

理论

从理论上讲，我们将故障排查过程定义为反复采用假设-排除手段的过程：[24]针对某系统的一些观察结果和对该系统运行机制的理论认知，我们不断提出一个造成系统问题的假设，进而针对这些假设进行测试和排除。

如图12-1所示的理想模型中，我们从收到系统问题报告开始处理问题。通过观察系统的监测指标（telemetry）[25]和日志信息了解系统目前的状态。再结合我们对系统构建原理、运行机制，以及失败模型的了

解，提出一些可能的失败原因。

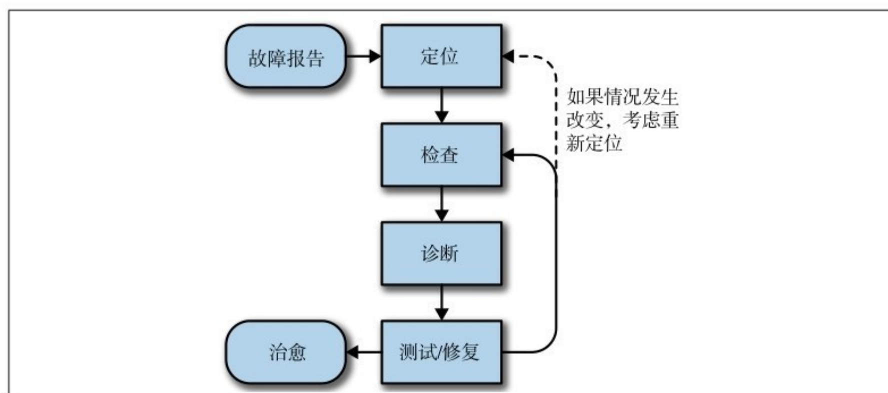


图12-1：通用的故障排查流程

接下来，我们可以用以下两种方式测试假设是否成立。首先，可以将我们的假设与观察到的系统状态进行对比，从中找出支持假设或者不支持假设的证据。或者，我们可以主动尝试“治疗”该系统，也就是对系统进行可控的调整，然后再观察操作的结果。第二种方式可以让我们更好地理解系统目前的状态，以及造成系统问题的可能原因。用上述两种方式中的任意一种，都可以重复测试假设，直到找到根本原因。这时，就可以采取纠正措施修复问题，防止问题重现，并且书写事后报告。当然，在找到根本原因之前，也可以采取措施修复一些表面问题。

常见的陷阱

造成低效的故障排查过程的原因通常集中在定位（triage）、检查和诊断环节上，主要由于对系统不够了解而导致。下面列举了一系列常见的陷阱，读者应该小心避免：

- 关注了错误的系统现象，或者错误地理解了系统现象的含义。这样会在错误的方向上浪费时间。

- 不能正确修改系统的配置信息、输入信息或者系统运行环境，造成不能安全和有效地测试假设。

- 将问题过早地归结为极为不可能的因素（例如认为是宇宙射线造成数据变化，虽然有可能发生，但是并不应该在解决问题初期做这个假

设），或者念念不忘之前曾经发生过的系统问题，认为一旦发生过一次，就有可能再次发生。

- 试图解决与当前系统问题相关的一些问题，却没有认识到这些其实只是巧合，或者这些问题其实是由于当前系统的问题造成的。（比如发现数据库压力大的情况下，环境温度也有所上升，于是试图解决环境温度问题。）

要避免第1点和第2点，需要更详细地学习系统的运行原理，同时了解分布式系统运行的基本模式。要避免第3点，需要记住，不是所有的失败情况的出现概率都相同　就像谚语中说的“当你听到蹄子声响时，应该先想到马，而不是斑马。”[26]而且尤其要注意的是，当所有的可能都存在的时候，我们应该优先考虑最简单的解释。[27]

最后，我们要记住，相关性（correlation）不等于因果关系（causation）。[28]一些同时出现的现象，例如集群中的网络丢包现象和硬盘损坏现象可能是由同一个原因造成的，比如说供电故障。但是网络丢包现象并不是造成硬盘损坏现象的原因，反之亦然。更糟的是，随着系统部署规模的不断增加，复杂性也在不断增加，监控指标越来越多。不可避免的，纯属巧合，一些现象会和另外一些现象几乎同时发生。[29]

理解我们逻辑推理过程中的错误是避免这些问题发生的第一步，这样才能更有效地解决问题。区分我们知道什么，我们不知道什么，我们还需要知道什么可以让查找问题原因和修复问题更容易。

实践

在实践中，故障排查过程当然不一定和理想情况下的模型完全一致。有一些简单的步骤可以使处理问题的整个过程更容易，以及更有效。

故障报告

每个系统故障都起源于一份故障报告，可以由自动报警产生，或者仅仅是你的同事说“系统很慢”。有效的故障报告应该写清预期是什么，

实际的结果是什么，以及如何重现。[30]理想情况下，这些报告应该采用一致的格式，存储在一个可以搜索的系统中，例如Bug记录系统中。很多团队都使用定制表单，或者小型的Web收集信息程序，同时自动发送和传递错误报告。还可以为常见问题提供一个自服务分析工具或者自服务修复工具，这也可以帮助错误汇报。

在Google，为每个错误报告提交一个Bug是常见做法，包括由E-mail和IM收到的错误报告。这样做的好处是保证每个问题都有调查历史和解决方案，可供未来引用。很多团队都不鼓励将问题直接汇报给某个具体的人：这样需要额外步骤将错误报告转化成Bug，经常产生低质量报告。而且容易导致解决的压力集中在几个问题汇报人熟悉的团队成员身上，而不是目前值班的成员（参见第29章）。

莎士比亚搜索服务出问题了！

在为莎士比亚搜索服务on-call的值班过程中，你收到了一个报警。ShakespeareBlackboxProbe_SearchFailure：你的黑盒监控系统在过去5分钟内搜索“the forms of things unknown”无法得到正确结果。报警系统自动创建了一个Bug，同时自动填入了黑盒探针结果的URL和on-call手册的相关链接，并且将这个Bug指派给了你，行动开始了！

定位

当你收到一个错误报告时，接下来的步骤是弄明白如何处理它。问题的严重程度大不相同，有的问题只会影响特定用户在特定条件下的情况（可能还有临时解决方案），而有的问题代表了全球范围内某项服务的不可用。你的反应应该正确反映问题的危害程度：对大型问题，立即声明一个全员参与的紧急情况可能是合理的（参见第14章），但是对小型问题就不合适了。合理判定一个问题的严重程度需要良好的工程师判断力，同时，也需要一定程度的冷静。

在大型问题中，你的第一反应可能是立即开始故障排查过程，试图尽快找到问题根源。这是错误的！不要这样做。

正确的做法应该是：尽最大可能让系统恢复服务。这可能需要一些

应急措施，比如，将用户流量从问题集群导向其他还在正常工作的集群，或者将流量彻底抛弃以避免连锁过载问题，或者关闭系统的某些功能以降低负载。缓解系统问题应该是你的第一要务。在寻找问题根源的时候，不能使用系统的用户并没有得到任何帮助。当然，快速定位问题时仍应该及时保存问题现场，比如服务日志等，以便后续进行问题根源分析时使用。

在初级飞行员的课程中讲到，在紧急情况中，飞行员的首要任务是保持飞机飞行（参见文献[Gaw09]）。相比保证乘客与飞机安全着陆，故障定位和排除是次要目标。这种方法也同样适用于计算机系统：如果一个Bug有可能导致不可恢复的数据损坏，停止整个系统要比让系统继续运行更好。

对新SRE来说，这个想法是反直觉，令人不安的。对以前曾经有过产品研发背景的人来说更有点难以接受。

检查

我们必须能够检查系统中每个组件的工作状态，以便了解整个系统是不是在正常工作。

在理想情况下，监控系统记录了整个系统的监控指标，如第10章中论述的那样。这些监控指标是找到问题所在的开始。查看和操作time-series图表是理解某个系统组件工作情况的好办法，可以通过几个图表的相关性确定问题根源。[31]

日志是另外一个无价之宝。在日志中记录每个操作的信息和对应的系统状态可以让你了解在某一时刻整个组件究竟在做什么。一些跟踪工具，例如Dapper（参见文献[Sig10]）提供了非常有用的了解分布式系统工作情况的一种方式。但是不同的产品需要极为不同的跟踪系统的设计（参见文献[Sam14]）。

日志

文本日志对实时调试非常有用。将日志记录为结构化的二进制文件通常可以保存更多信息，有助于利用一些工具进行事后分析。

在日志中支持多级记录是很重要的，尤其是可以在线动态调整日志级别。这项功能可以让你在不重启进程的情况下详细检查某些或者全部操作，同时这项功能还允许

当系统正常运行时，将系统日志级别还原。根据服务的流量大小，有可能采用采样记录会更好，例如每1000次操作记录一次。

更进一步，你可能需要在日志系统中支持过滤条件——记录所有满足X的操作。X的范围很广，例如过滤 Set RPC 中内容长度小于1024字节的，或者是操作时间超过10 m s的，或者是调用了 rpc_handler.py中doSomethingInteresting（）函数的操作的。日志记录系统最好能够按需、快速、有选择启用。

暴露目前的系统状态是第三个重要工具。比如，Google软件服务器包含一系列监控页面，可以显示最近接收的RPC采样信息。这样我们可以直接了解该软件服务器正在与哪些机器通信，而不必去查阅具体的架构文档。

这些监控页面同时显示了每种类型的RPC错误率和延迟的直方图，这样可以快速查看哪些RPC存在问题。有的系统的监控页面中显示了系统目前的配置文件信息，或者提供了查询数据的接口。例如，Google的Borgmon服务（参见第10章），可以显示出当前使用的规则文件列表，甚至允许单步跟踪某一条规则的计算过程。

最后，你可能需要使用一个该系统的真实客户端，以便了解这个组件在收到请求后具体返回了什么信息。

调试莎士比亚搜索服务

使用Bug中的黑盒监控系统结果链接，你会发现探针程序给/api/search发送了一个HTTP GET请求：

```
{
    'search_text': 'the forms of things unknown'
}
```

探针程序希望得到一个HTTP 200返回值，以及一个JSON结果集，

精确匹配为：

```
[{
        "work": "A Midsummer Night's Dream",
        "act": 5,
        "scene": 1,
        "line": 2526,
        "speaker": "Theseus"
}]
```

探针程序被设置为每分钟探测一次，在过去10分钟内，大概一半的探测请求成功了，但是并没有任何明显的分布模式。不幸的是，探针程序没有告诉你当探测失败时，具体返回了什么结果，所以你将这个问题记录了下来，以便日后修复。

使用`curl`，你手动发送了一个请求，获得了一个失败回复：HTTP 502（Bad Gateway），也没有任何结果信息。同时，结果中包含了一个HTTP头，X-RequestTrace，其中写明了处理请求的后端服务器地址。利用这个信息，可以仔细检查这些后端服务器是否工作正常。

诊断

对系统设计的详细了解，是针对目前系统出现的问题提出合理猜想的主要帮助。但是，这里有一些通用的手段可以在没有领域知识的情况下提供帮助。

简化和缩略

理想情况下，系统中的每个组件都应该有明确定义的接口，并且每个接口都按照固定规则将输入转化为输出（在我们的例子中，该组件接收搜索文字请求，并且返回符合条件的结果）。我们就可以通过检查组件之间的链接，或者是中间传输的数据来判断某个组件是否正常工作。将已知的测试数据输入到系统中，检查输出是否正确（这就是黑盒测试的一种），是非常有帮助的。尤其是使用针对某种错误情况的专门测试数据。如果系统有配套的测试用例，那么调试起来就会很容易。甚至这套测试用例可以用于非生产环境，非生产环境通常可以执行更具有侵入性和危害性的操作。

问题分解（Divide & Conquer）也是一个非常有用的通用解决方案。在一个多层系统中，整套系统需要多层组件共同协作完成。最好的办法通常是从系统的一端开始，逐个检查每一个组件，直到系统最底层。这样的策略非常适用于数据处理流水线。在大型系统中，逐个检查可能太慢了，可以采用对分法（bisection）将系统分为两部分，确认问题所在再重复进行。

“什么”“哪里”和“为什么”

在一个异常系统中，该系统经常还正在执行某种操作，只是这些操作不是你想让系统执行的操作。那么找出系统目前正在执行“什么”，然后通过询问该系统“为什么”正在做这些操作，以及系统的资源都被用在了“哪里”可以帮助你了解系统为什么出错。[32]

对问题现象的解析

现象：一个Spanner集群出现延迟过高的情况，RPC请求超时。

为什么：Spanner 服务器的任务实例把CPU占满了，无法处理用户发来的请求。

哪里：Spanner服务器的CPU消耗在了哪里？通过采样（profiling）得知该任务正在将日志记录排序，写入磁盘。

哪里：这段排序代码中的哪部分在消耗资源？是在一段正则表达式的计算过程中。

解决方案：重写该正则表达式，避免使用回溯（backtracking）。在代码中寻找类似的问题。考虑使用RE2，该类库保证不使用回溯，同时保障运行时间与输入呈线性关系。[33]

最后一个修改

计算机系统有惯性存在：我们发现，一个正常工作的计算机系统会维持工作，直到某种外力因素出现，例如一个配置文件的修改，用户流量的改变等。检查最近对系统的修改可能对查找问题根源很有帮助。[34]

设计良好的系统应该有详尽的生产日志，记录整个架构中新版本部署或者配置文件的更新。这包括从处理用户请求的服务进程，一直到集群中每个节点的安装的版本信息。通过将系统的环境改变与系统性能和行为进行对比，可能比较有用。例如，在服务的监控页面上，你可以将系统的错误率图表和新版本的部署起始时间和结束时间标记在一起，如图12-2所示。

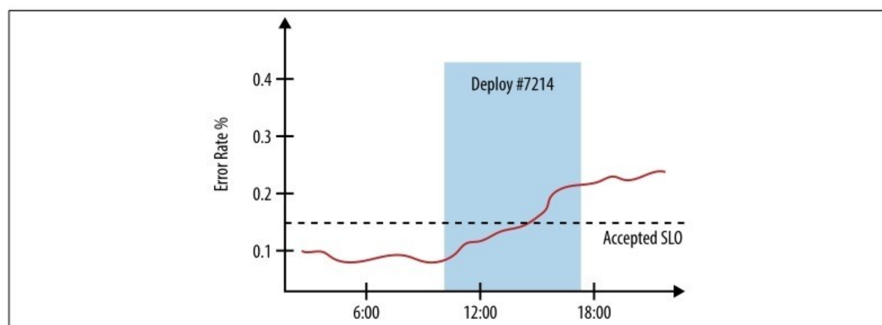


图12-2：错误率图表以及部署开始时间和结束时间

通过手动向/api/search发送一个请求（参见本章前面“调试莎士比亚搜索服务”材料），并且观察到错误回复中包含处理请求的后端服务器，可以让你排除掉API前端服务器和负载均衡服务出现问题的几率：因为如果请求没有到达后端服务器，那么就不会带有这个信息。现在你可以将精力集中在后端服务器上，通过分析日志，发送测试请求来测试，同时检查后端服务的监控指标。

有针对性地进行诊断

虽然前面论述的通用性工具在很多问题上都很有用，但是针对具体系统开发的诊断工具和诊断系统会更有用。Google SRE花费了很多时间开发这样的工具。很多工具只适用于某个特定系统，但是在不同的团队和系统之间应该寻找共同点，以减少重复劳动。

测试和修复

有了一个相对较短的可能原因列表，接下来就应该试着找出具体哪个原因才是真正的根源问题。通过执行一些具体的试验，可以确认和推翻我们所列举的假设。举例来说，如果认为一个错误是由于应用逻辑服

务器和数据库之间的网络连接问题导致的，或者是由于数据库拒绝连接导致的。通过测试应用服务器的用户名和密码实际链接数据库可以验证第二个假设。通过ping数据库服务器可以测试第一个假设，但是这取决于具体的网络拓扑环境、防火墙配置等。通过查看源代码，并且试图模拟源代码执行，也可以看出哪里出现了错误。

在设计测试时，有一些考量必须时刻记住。（这些测试可能是简单的ping测试，或者是复杂的测试，例如将用户流量导出一个集群，同时使用特殊构造的请求试图发现资源竞争问题。）

- 一个理想的测试应该具有互斥性，通过执行这个测试，可以将一组假设推翻，同时确认另外一组假设。在实际执行中，这比较难。

- 先测试最可能的情况：按照可能发生的顺序执行测试，同时考虑该测试对系统的危险性。先测试网络连通性，再检查是否是最近的配置文件改变导致用户无法访问某机器可能更合理。

- 某项测试可能产生有误导性的结果。例如：防火墙规则可能只允许某些特定IP访问，所以在你的工作机上ping 数据库可能会失败，而实际从应用服务器上ping数据库可能是成功的。

- 执行测试可能会带来副作用。举例说明：让一个进程使用更多CPU可能会让某些操作更快，也可能会导致数据竞争问题更容易发生（单线程与多线程运行）。同样的，在运行中开启详细日志可能会使延迟问题变得更糟，同时也让你的测试结果难以理解：是问题变得更加严重了，还是因为开启详细日志的原因？

- 某些测试无法得出准确的结论，只是建议性的。死锁和数据竞争问题可能是非常难以重现的，所以有的时候你并不能找到非常确切的证据。

将你的想法明确地记录下来，包括你执行了哪些测试，以及结果是什么。[35]尤其是当你处理更加复杂的问题时，良好的文档可以让你记住曾经发生过什么，可避免重复执行。[36]如果你修改了线上系统，例如给某个进程增加了可用资源。系统化和文档化这些改变有助于将系统

还原到测试前的状态，而不是一直运行在这种未知状态下。

神奇的负面结果

作者：**Randall Bosetti**

编辑：**Joan Wendt**

“负面结果”指一项试验中预期结果没有出现，也就是该试验没有成功。这里的试验是广义范围的，包括新的系统设计、启发性算法或工作流程没有改进他们试图取代的旧系统。

负面结果不应该被忽略，或者被轻视。意识到自己的错误通常意义更大：一个明确的负面结果可以给某些最难的设计问题画上句号。一个团队经常同时有两个看起来可行的设计方向可供选择，选择任意一个方向都需要回答有关是否另外一个方向更好的问题。该问题通常是模糊而抽象的。

一项试验中出现的负面结果是确凿的。这些结果确切地告诉了我们有关生产环境中的信息、设计理念对错或者现有系统的性能极限。这些负面结果能够帮助其他人决定他们的试验和设计是否值得一试。举例来说，某个研发团队可能决定不使用某个Web服务器，因为由于锁竞争的缘故只能处理800个并发连接，而不是需要的8000个。当下一个研发团队想要评测Web服务器时，他们可以直接利用之前的负面结果判断而不是从头开始。他们可以根据（a）需要的并发连接数少于800个（b）锁竞争问题已经被解决了来决定是否使用。

就算该负面结果无法被其他人直接利用，这项试验收集到的数据也可以帮助其他人选择新的试验，或者避免之前设计中的问题。微型性能测试、文档化的反模式，以及某项目的事后报告都属于这个范畴。我们在设计实验的时候应该将可能的负面结果考虑在内，因为一个可靠的、应用广泛的负面结果对其他人更有帮助。

工具和方法可能超越目前的试验，为未来的工作提供帮助。例如，性能测试工具和负载生成器从成功或失败的试验过程中都可以产生。很多Web服务器管理人员都从Apache Bench，一个负载测试软件中获利。

虽然第一个测试结果很可能是不尽如人意的。

为了可重复地试验而构建的工具可能还存在间接好处：虽然目前构建的一个应用程序可能并不会因为将数据库运行在SSD上，或者建立更好的索引而更快。但是下一个应用程序可能会。将这些测试写成脚本有助于保证在下一个项目中你不会忘记和错过好的优化点。

公布负面结果有助于提升整个行业的数据驱动风气。在我们的数据中记录负面结果和其他非统计显著的结果有助于降低数据的偏差度，同时为其他人做出了接受未知可能性的榜样。公布所有数据鼓励了其他人也这么做，从而使得整个行业的学习速度变快了。SRE已经通过高质量的事后报告体会到了这一点，这些报告极大地促进了生产环境稳定性的提升。

公布结果。如果你对一项测试的结果感兴趣，那么很有可能其他人也感兴趣。当你公布结果的时候，其他人不需要再重新设计和运行一套类似试验。不公布负面结果是很常见的，因为他很容易被理解为试验“失败了”。有一些试验根本就不可能成功，而发现他们的最好方法就是通过评审。更多的试验压根没有公布数据，因为很多人错误地认为负面结果意味着没有价值。

每个人都应该主动公布自己已经排除的设计、算法和团队工作流程等。鼓励你的同行们意识到，负面结果是有计划的冒险行为，每个设计良好的试验都是有价值的。应对任何没有提到失败的设计文档、性能评估文档以及短文保持怀疑，因为这篇文章可能经过了过度筛选，也可能因为作者的方法不是那么严谨。

最后，公开你自己觉得意外的结果会让其他人—包括未来的你不会再次感到意外。

治愈

理想状况下，现在你已经将一系列可能的错误原因减少到了一个。下一步，我们想要证明这就是问题根源。在生产环境中试图通过重现一个原因而明确地证明它就是问题的原因是很困难的。因为如下几个原

因，我们经常只能发现可能的原因。

- 系统复杂度。很有可能有多个因素共同作用导致系统问题。[37]真实系统通常是路径依赖的，也就是说，它们必须处于一个特定状态下问题才会发生。

- 在生产环境中重现某个问题也许是不可能的。要么因为将系统置于某种问题状态过于复杂，或者是系统过于重要，不能再出问题。如果有一套非生产环境的复制系统可能好一些，但是需要额外付出一定的成本。

当你最终确定了某个因素是问题根源时，应该将系统中出错的部分，你是如何定位问题的，和你是如何修复问题的，如何防止再次发生等写下来。这就是事后总结。希望这时候系统是活着的！（事后总结也称为验尸报告，但是这里是在问题解决后才写的，服务已经恢复。）

案例分析

App Engine，[38]是Google 云计算平台的一部分。App Engine是一个PaaS产品，开发者可以利用这个产品用Google的基础架构设施构建自己的服务。有一个内部客户提交了一份问题报告，声称他们最近观察到了系统延迟、CPU使用率以及运行进程的数量大幅增长。他们的服务是一个给开发者使用的内容管理系统（CMS）。[39]该内部客户没有找到最近任何的程序改动会导致资源使用的增多，同时该App也没有用户流量的增长（参见图12-3），所以他们想知道这是否是App Engine服务本身造成的。

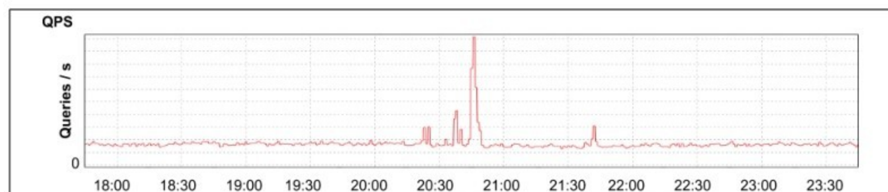


图12-3：应用程序每秒接收到的请求，显示出一个短暂的峰值，接下来恢复正常。

我们的调查显示，该程序延迟的确上升了一个数量级，如图12-4所示。与此同时，CPU时间（参见图12-5）和服务进程的数量（参见图

12-6）几乎成四次方增长。很明显某些东西有问题，现在是故障排查的时间了！

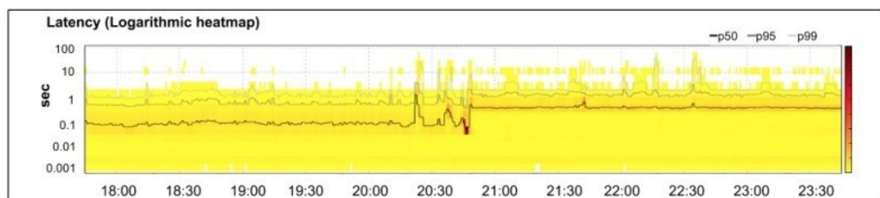


图12-4：应用程序延迟图，显示出50%、95%、99%请求的延迟（以线表示），同时用热力图显示了在指定时间内请求的分布情况。

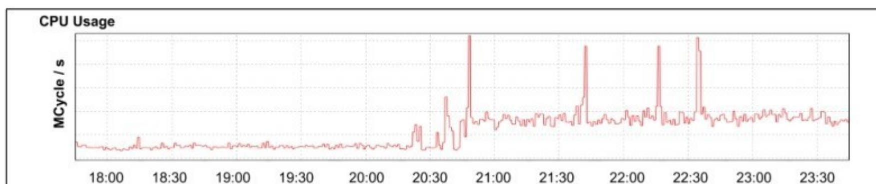


图12-5：应用程序CPU使用率汇总

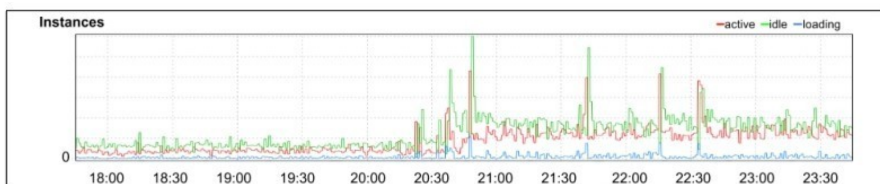


图12-6：应用程序运行的进程数量

通常来说，延迟和资源使用的突然增加意味着流量的突然增加，或者系统配置的改变。然而，我们可以轻易地将这两个因素排除在外：虽然App在20:45 接收到的峰值流量可以解释资源的临时增加，但是我们认为随着流量恢复，资源使用率也应该回到正常水平。这次峰值显然不应该从App开发者提交错误报告和我们开始调查时算起持续数天之久。第二，App性能的改变发生在星期六，当时没有App的改动，也没有生产环境的变更在执行。App Engine服务最近的代码改动和配置改动在数天前已经完成。再次，如果这个问题是由服务产生的，我们预计将会在其他App中（运行在同样的架构下）看到类似的情况。然而，没有其他App出现类似情况。

我们将这个问题汇报给了App Engine的开发团队，让他们协助调查

是否该客户遇到了服务基础架构中的某些特质。开发团队也无法找到任何可疑点。然而，有一个开发者发现了延迟上升和某个具体数据存储API的用量（merge_join）增多的相关性，这通常代表了在从数据库中读取数据时出现了索引问题。为这个App使用的数据属性增加一个综合索引可以加速这些查询，理论上来说，可以加速整个App。但是我们需要找到具体哪个属性需要这个索引。快速浏览了一下App的代码也没有提供明显的疑点。

这时，我们拿出了工具集中的重型工具：Dapper（参见文献[Sig10]），它可以跟踪单个HTTP请求的具体步骤。从前端的接收、反向代理到App代码返回一个结果，我们可以跟踪每个相关服务器发出的RPC。通过这些信息，在发往数据服务的RPC请求中最终找到了需要添加索引的属性，最后增加了相关的索引。

在调查过程中，我们同时发现一些针对静态资源的访问，例如图片，并非通过数据服务提供，也非常慢。在查看文件级别的图表时，我们发现这些请求在几天前一直很快。这同时暗示了我们观察到的merge_join和延迟上升只是伪相关，关于不正确的索引使用造成延迟上升的理论是完全错误的。

通过对静态资源请求缓慢的检查发现，从应用程序发出的大部分RPC是到内存缓存服务（memcache），所以请求应该非常快，在毫秒级。这些请求的确非常快，所以问题并不是出在这里。然而，在App接收到请求到发出第一个RPC时，有250ms的时间App在做某件未知的事。因为App Engine运行的是用户提供的代码，SRE团队并不会检查这些代码，所以我们不知道这段时间App在做什么。同样，Dapper也不能帮助我们，因为它只能跟踪RPC请求，而这段时间App没有发送任何RPC。

基于目前得到的信息，问题仍然是一个谜，我们决定暂时不解决它。因为用户下下周已经计划了公开上线，而我们并不知道多长时间才能找到和修复这个问题。我们建议用户增加App的资源，同时选择CPU资源最充足的实例类型。通过这些措施，App延迟降低到可接受的范围，虽然不是最佳情况。我们认为这样的方法已经足够好，可以让用户

的App成功上线，而我们可以放松地调查问题。[40]

在这时，我们怀疑这个App是另外一个延迟和资源使用变化因素的受害者：工作负载类型的改变。在App延迟改变之前，我们看到了App对数据服务的写请求有增加。但是因为这个增加幅度并不是太大，同时也不持久，我们很早就认为这只是偶然。但是这种行为模式的确很像一种常见模式：App的一个实例使用从数据服务读取的数据初始化，然后将这些数据存储在内存中。这样这个实例可以避免反复读取不经常改变的配置文件，而只是做内存检查。然而，处理每个请求的时间经常与配置文件的数量同步增长。[41]我们无法证明这就是问题根源，但是这是一个常见的反模式。

App开发者在代码中增加了一些标记，帮助理解App的具体执行时间。他们找到了每个请求都会调用的一个具体方法，该方法检查用户是否有权限访问某个路径。这个方法为了最大程度减少数据服务和内存缓存服务的访问数量使用了一个缓存层，该缓存层将白名单中的物件存放于内存中。正如一个App开发者在调查过程中记录的那样："虽然我现在还不知道具体着火点在哪里，但是白名单缓存层中的滚滚浓烟已经要把我弄瞎了。"

一段时间后，问题根源找到了：App的权限检查系统中有一个存在时间很长的Bug，每当一个路径被访问时，一个新的白名单物件就被创建出来，同时保存在数据库中。在公开上线之前，一个自动化安全扫描程序被用来扫描该App是否有漏洞，由此造成的副作用就是该扫描过程在半小时内产生了几千个白名单物件。这些无用的白名单物件必须被每个请求都检查一次，由此导致了延迟的上升。这中间没有触发任何RPC请求。修复Bug，删除了无用的白名单物件之后，App性能恢复到了正常水平。

使故障排查更简单

有很多方法可以简化和加速故障排查过程。可能最基本的是：

- 增加可观察性。在实现之初就给每个组件增加白盒监控指标和结构化日志。

- 利用成熟的、观察性好的组件接口设计系统。

保证这些信息用一个一致的方法在整个系统内传递，例如：使用唯一标识标记所有组件产生的所有相关RPC。这有效地降低了需要对应上游某条日志记录与下游组件某条日志记录的要求，加速了检查和恢复。

代码中对现存错误的假设，以及环境改变也经常会导致需要故障排查过程。简化、控制，以及记录这些改变可以降低实际故障排查的需要，也能让故障排查更简单。

小结

我们讨论了能够将故障排查过程变得简单和可理解的步骤，这样新手也可以有效解决问题。对故障排查采用系统化的手段，而不是依靠运气和经验，将有助于限定你的服务的故障恢复时间（MTTR），从而为你的用户提供更好的使用体验。

第13章 紧急事件响应

作者：**Corey Adam Baye**

编辑：**Diane Bates**

东西早晚要坏的，这就是生活。

不管一个组织有多大，做的事情有多么重要，它最明显的特质就是：在紧急事件来临时人们如何应对。没有几个人天生就能很好地处理紧急情况。在紧急情况下恰当处理需要平时不断地进行实战训练。建立和维护一套完备的训练和演习流程需要公司董事会和管理层的支持，同时需要一批专注投入的人。要想创造一个人们可以依赖的环境，可以用合理的资源有效地应对紧急情况，这些元素都是不可或缺的。

第15章讨论了如何通过书写事后总结，将紧急事件的处理过程变成一个学习机会，本章为此提供了更多实际案例。

当系统出现问题时怎么办

首先，别惊慌失措！这不是世界末日，你也并不是一个人在战斗！作为一个专业人士，你已经接受过如何正确处理这样的情况的训练。通常来说，我们处理的问题一般不涉及真实的物理危险，只是计算机系统出现了问题。最差的情况下，也只是半个互联网都停止运转了（Google网络规模已经是全球排名前三）。所以请深吸一口气，慢慢来。

如果你感到自己难以应对，就去找更多人参与进来。有的时候，甚至需要将整个公司的人全部加入进来。如果你的公司已经建立了一套灾难应急流程（参见第14章），请确保你对这个流程非常熟悉，并且按照流程行事。

测试导致的紧急事故

Google经常主动进行灾难处理和应急响应演习（参见文献[Kir12]）。SRE故意破坏系统、模拟事故，然后针对失败模式进行预防以提高可靠性。大多数时候，这些模拟事故都能按照计划进行，目标测试系统和依赖系统基本都能正常工作。我们利用这些测试曾经发现过一些系统的弱点和潜在的依赖关系，制定了修改计划以解决这些问题。但是有时候，我们的假设和实际结果相差很远。

下面是一个具体的例子，在这次测试中，我们发现了一系列意外的系统依赖。

细节

我们想要找到依赖我们大型分布式MySQL集群中的一个测试数据库。测试计划是通过限制访问权限来屏蔽一百多个数据库中的一个。没有任何一个人能想到接下来发生的事。

响应

测试开始的仅仅几分钟后，大批服务汇报公司内部和外部用户都无法访问某个关键业务系统。这些关键系统业务时通时断，或者只是部分可用。

SRE自然认为这些问题是测试导致的，所以立即终止了这项测试。我们试着通过回滚来恢复访问权限，但是意外地发现这项操作没有成功。SRE没有惊慌失措，而是立刻集思广益找出了如何通过其他手段恢复访问权限。用一个以前已经测试过的方案，我们成功地在集群副本和灾备系统上恢复了访问权限。与此同时，我们联系上了负责开发数据库连接类库的开发者，修复了代码中的问题。

在灾难发生的一小时内，所有访问权限都得到了恢复，同时所有的服务都恢复了正常。这次测试带来的严重后果促使开发者对代码类库中的问题进行了一次彻底的修复，同时制定了一个周期性测试机制来保证这类严重问题不再重现。（此项事故应该是由于数据库连接类库中没有正确处理某个数据库失去响应的情况，导致了进程阻塞或者其他所有数据库也不能被正常访问）。

事后总结

做得好的地方

受到这次事故影响的服务及时地在公司内部进行了沟通。我们有足够的信息假设是我们的这次可控测试导致了事故，从而立即停止了该项测试。

我们成功地在一个小时内（收到第一个事故报告起）恢复了服务。有些受影响的团队采用了另外一种策略，重新配置了他们的服务，将测试数据库从配置中去掉。这些并行的努力使整个服务更快地得到了恢复。

事故发生之后产生的待办事项（指修复代码工作）也很快得到了快速而彻底地解决，从而避免了类似的事故再次发生。同时我们制定了周期性的测试，保证类似的问题不会再次发生。

做得不好的地方

虽然这项测试在进行之前进行了充分评审，评审中认为这项测试是很合理的，事实证明我们对相关系统的了解还不够。

我们没有正确地遵守应急响应流程，因为这个流程几个星期前刚刚建立，还没深入人心。这项流程本来可以保证所有的相关服务和用户更好地了解整个事故的全过程。为了避免类似情况的发生，SRE将继续优化和测试应急响应流程工具，确保未来应急事故中的所有人都十分了解该流程。

由于我们没有在测试环境中测试回滚机制，没有发现这些机制其实是无效的，导致了事故总时长被延长了。我们现在要求在大型测试中一定先测试回滚机制。

变更部署带来的紧急事故

正如你想象的那样，Google有很多配置文件。这些文件非常复杂，同时我们在不停地进行修改。为了避免配置文件导致意外事故的发生，我们在部署配置文件之前会运行大量的测试。但是由于Google基础设施的规模和复杂程度，不可能完全预料每种情况，有的时候配置文件改变并不能完全按照计划进行。

下面就是一个例子。

细节

在某个星期五，一个防滥用基础服务（anti-abuse）的新版配置文件被推送到所有的服务器上。这个服务和Google所有对外服务都有直接联系。这个新版配置文件触发了一个Bug，导致这些对外服务都进入了崩溃循环（crash-loop）。因为Google内部很多基础服务也依赖于这些内部服务，所以导致很多内部应用程序也失去了响应。

事故响应

几秒内，各种监控就开始报警，声称某些网站失去了响应。一些on-call工程师同时发现公司内网好像出现了问题，于是他们都前往一个专门的灾难安全屋（panic room）。这个房间中有Google生产环境的专线连接。他们意外地发现，越来越多的人因为网络问题来到了这里。

在配置文件发布的5分钟后，负责发布这个配置的工程师也发现公

司内网出现了问题，但是还没意识到更大范围的生产系统问题。该工程师发布了一份修正过的配置文件，以便将之前的问题回滚。此时，各项服务逐渐开始恢复。

在第一次发布的10分钟后，on-call工程师按照内部应急流程宣告进入紧急状态。他们开始通知公司内的其他部门目前的情况。负责发布配置的工程师告诉on-call工程师这次事故可能是由于他发布的配置文件造成的，而且目前已经回滚了。但是，有些服务遇到了由于最初的事故导致的另外的Bug或者错误的配置等影响，一个小时后才能恢复。

事后总结

做得好的地方

以下几个因素使得很多Google内部系统及时恢复了。

首先，监控系统及时检测和汇报了问题。但是这里应该记录下来，我们的监控系统还是存在问题：报警信息不停地重复报警，让on-call工程师难以应对，导致在正常的和紧急的通信通道中充斥了大量垃圾信息。

问题被检测到之后，应急响应流程处理得当，其他人得到了清晰和及时的事态更新。我们的带外通信系统（out-of-band，指专线数据中心连接）保证了在复杂软件问题下，每个人都能保持连接。这次经历提醒了我们，SRE为什么要保持一些非常可靠的、低成本备份访问系统运行，因为我们在这种情况下就会用到！

在这些带外通信系统之外，Google还有命令行工具和其他的访问方式确保我们能够在其他条件无法访问的时候进行更新和变更回滚。这些工具和访问方式在这次事故中起到了重大作用，但是工程师应该更加频繁地测试，以便更为熟悉它们。

Google的系统架构体系提供了另外一层保护，这次受影响的系统包含了限速机制，限制他们给新客户端分发配置变更的速度。这些限速措施可能抑制了崩溃循环的速度，使得某些任务可以在崩溃之前还可以处理一些服务器请求，从而避免了进入彻底不可用的状态。

最后，我们不应该小看运气这个因素。这次我们能够迅速解决问题是因为进行部署变更的工程师恰好在监控某些实时通信频道，这并不是发布过程中要求的。正因为如此，这个工程师注意到在变更推送结束后，频道里出现了大量的公司网络访问故障报告，从而迅速回退了该配置。如果配置没有这么快地被回退，这项事故可能会持续相当长的时间，同时变得更难解决。

我们从中学到的

在之前的部署中，导致问题的这项新功能经历了一个完整的部署试验（Canary）周期却没有触发该Bug，因为在那次测试中配置文件中没有用到一个非常特别、很少用到的关键词。这次触发问题的变更，并没有被认为是非常危险的，导致这项改动仅仅经历了一个简单的部署试验过程。当这次变更在全球部署的时候，这个没有经过测试的关键词/新功能组合导致了灾难的发生。

具有讽刺意味的是，我们本来计划在下个季度提高部署测试流程和自动化的优先级。这次事故的发生直接将他们的优先级提高了，同时强调不管风险看起来有多小，也要经过严格完整的部署测试。

正如我们所料，监控系统在这次灾难中发出了很多报警，因为全球每个节点都失去响应数分钟。这严重干扰了on-call工程师的工作，同时让在这次事故的参与者们沟通起来更困难了。

Google 大量使用自研工具。我们大部分的在线调试与内部沟通工具都依赖这次进入崩溃循环的任务。如果这次的故障没能很快解决的话，我们的在线调试能力将受到极大的影响。

流程导致的严重事故

我们在机器管理自动化上投入了大量的时间和精力，以使我们能很轻松地在整个集群里运行、停止和重新配置大量任务。但是当意外来临时，自动化的效率有时可能是很可怕的。

下面这个例子讲述了有的时候动作太快不一定是好事。

细节

在一项常规自动化测试中，该测试针对同一个集群（马上即将退役的）发送了两个连续的下线请求（turndown）。在处理第二个下线请求时，自动化系统中的一个非常隐蔽的Bug将全球所有数据中心的所有机器加到了磁盘销毁（diskerase）的队列中，这导致硬盘数据被清空。细节请参看第7章的“自动化：允许大规模故障发生”材料。

灾难响应

在第二个下线请求提交不久之后，on-call工程师收到一个警报，声称第一个小型数据中心的全部机器都被下线了，即将退役。工程师的调查显示，这些机器已经被转移到了磁盘销毁队列中。所以按照正常流程，on-call工程师将用户流量导向了其他地区（drain）。因为这些机器的硬盘已经被清空了，为了避免这些请求失败，on-call工程师将用户流量导向了其他正常工作的地区。

不久之后，全球各地数据中心都发出了报警。on-call工程师收到报警后，将整个团队的自动化工具全部停止，避免问题进一步发展。随后，他们停止或者暂停了更多的自动化工具，以及常规的生产环境维护活动。

在一个小时之内，所有用户流量都被导向其他的地方了。虽然用户可能面临延迟升高等问题，但是他们的请求还是能够被正常处理的。这次事故总算结束了。

现在最难的部分开始了：恢复。有些网络连接汇报堵塞得非常严重，所以网络工程师在网络关键节点部署了一些缓解措施。在这些节点中的某个数据中心被选中为第一个浴火重生。三个小时后，经过几个工程师的努力，这个数据中心被成功重建了，可以再次接受用户请求了。

美国团队将工作交接给了他们的欧洲伙伴团队。SRE设计了一个利用手动步骤快速执行重建操作的计划。整个团队分为三部分，每部分负责整个手工重建计划中的一步。在三天内，大部分服务器都恢复了服务，其余没有恢复的都将在下个月或者下两个月内恢复。

做得好的地方

在大型数据中心中，反向代理（前文没有提到受影响的服务是什么，这里提到了是一种反向代理）的管理方式和小型数据中心的管理方式截然不同。所以这次事故没有影响大型数据中心。on-call工程师能够迅速地将用户流量从小型数据中心导向大型数据中心。按照设计，这些大型数据中心可以处理全部流量。但是一些网络连接出现了拥堵情况，因此需要网络工程师部署一些临时措施。为了降低对最终用户的影响，on-call工程师将出现堵塞的网络节点设置为他们的最高优先级。

小型数据中心的下线请求处理过程非常高效和完美。从开始到结束，不到一个小时的时间内大量的小型数据中心都被成功地下线以及进行安全的硬盘擦除。

虽然下线自动化迅速将对应的站点监控系统也干掉了，但是on-call工程师成功地将这些改动快速恢复。这一点对他们评估事故严重程度提供了帮助。

应急事故处理流程在本章提到的第一个事故出现之后很快成熟了起来，这些工程师很快很好地执行了这些流程。在事故中，整个公司的沟通和协作是一流的，多亏了应急事故管理系统和平时的训练。所有相关的团队都参与进来，共同帮助解决问题。

我们从中学到的

这次事故的根源在于自动化系统对它发出的命令缺乏合适的合理性检查。当自动化系统第二次运行，获取机柜列表时得到了一个空白回应。该自动化系统没有抛弃这个回复，而是在整个机器数据库（Google Machine Database 记录了生产系统中的全部机器列表）上运行了一个空白过滤，导致将数据库中所有的机器都转移到了磁盘擦除队列中。有的时候零就等于全部。机器数据库响应了这个请求，于是其他的下线工作流开始迅速执行起来。

这些机器的重新安装是非常缓慢而且不可靠的。这个问题主要由于

重装过程使用了小文件传输协议（TFTP），同时使用了最差的网络质量标签（QoS）。每台机器的BIOS系统没有很好地处理传输问题。[42]在不同的网卡上，BIOS要么进入了停止状态（halt），要么进入了一种重启循环（reboot cycle）状态。BIOS无法每次完成启动文件的传输，更导致了安装器（installer）的负载上升。on-call工程师成功地将安装请求优先级提高，同时利用自动化工具重启了这些卡住的机器。

机器重装系统基础服务不能处理几千台机器的同时安装操作。这个问题主要来源于一个性能问题导致该服务只能在每台工作机器（worker）上同时并行运行两个安装操作。这个问题同时也在传输文件时使用了错误的QoS设置，和错误的超时设置。而且该服务在没有被擦除的机器上也要强制重新安装内核。为了处理这些问题，on-call工程师将问题升级给了对应的开发工程师，他们很快将系统调节到了更好的性能水平上。

所有的问题都有解决方案

时间和经验一再证明，系统不但一定会出问题，而且会以没有人能够想到的方式出问题。Google学到的最关键的一课是，所有的问题都有对应的解决方案，虽然对一个面对着疯狂报警的工程师来说，它可能不是那么显而易见。如果你想不到解决办法，那么就在更大的范围内寻求帮助。找到更多团队成员，寻求更多的帮助，做你需要做的一切事情，但是要快。最高的优先级永远是将手头问题迅速解决。很多时候，触发这个事故的人对事故了解得最清楚，一定要充分利用这一点。

非常重要的是，一旦紧急事件过去之后，别忘了留出一些时间书写事后报告。

向过去学习，而不是重复它

为事故保留记录

没有什么比过去的事故记录是更好的学习资料了。历史就是学习其他人曾经犯的错误。在记录中，请一定要诚实，一定要事无巨细。尤其重要的是，提出关键的问题。时刻寻找如何能在战术及战略上避免这项

事故的发生。公布和维护事后报告，确保全公司的每个人都能从中学到你所学到的知识。

在事故结束后，确保自己和其他人切实完成事故中总结的待办事项。这样能够避免未来再次发生以同样的因素触发的同样的事故。一旦开始仔细学习过去的事故，我们就能更好地避免未来的事故。

提出那些大的，甚至不可能的问题：假如……

没有什么比现实更真实的测试了。我们应该提出一些大的、没有确切答案的问题。假如整栋大楼的电源坏了怎么办？假如网络设备被泡在半米深的水里怎么办？如果主数据中心突然下线了怎么办？如果有人入侵了你的Web服务器怎么办？你怎么处理？找谁联系？谁来付钱？有对应的应急计划吗？你知道如何应对吗？你知道你的系统会如何应对吗？如果上述所说正在发生，你能够立刻最小化灾难损失吗？坐在你旁边的人能做到同样的事吗？

鼓励主动测试

面对失败，理论和实践是两个完全不同的领域。直到你的系统真的失败的那一刻，你并不真的了解它，以及依赖它的系统，或者它的用户会如何应对。不要预设任何假设，也不要依赖任何没有经过测试的假设。你是希望这个系统在星期六凌晨两点钟，公司大部分同事都还在参加黑森林中的团建时出现故障，还是希望和最可靠和最聪明的同事在一起仔细监控着它们上周详细评审过的测试时出现故障呢？

小结

上文论述了三种不同的系统失败情况。虽然三个紧急情况触发方式不同，一个由主动测试导致，一个由配置文件改动触发，另外一个由自动化程序触发。对这些事件的处理有着共同的特点，响应者没有惊慌失措。他们在必要的时候引入了其他人的帮助。他们都研究和学习过以前的事故记录。事故过后，他们都将系统改善为能更好地处理同类故障。每次新的失败模式发生时，应急处理者都将这些模式记录了下来；依靠这些事后报告，他们帮助了其他的团队学习如何更好地进行故障排除，

以及加固他们的系统以避免类似事故发生。同时这些处理者也主动测试了他们的系统，这些测试保障了相关修改确实修复了根源问题，同时在事故发生之前提前发现了其他的系统弱点。

随着我们的系统不断发展，每次事故和测试都让系统和流程不断进步。虽然本章中的案例分析是Google特有的，但是这种对待紧急事故的模式可以应用在各种不同规模的组织架构上。

第14章 紧急事故管理

作者：**Andrew Stribblehill**[43]

编辑：**Kavita Guliani**

有效的紧急事故管理是控制事故影响和迅速恢复运营的关键因素。如果事先没有针对可能发生的紧急事故进行过演习，那么当事故发生时，一切管理理念都起不了作用。

本章详细描述了一次没有采用紧急事故处理流程而导致失控的事故，还描绘了如何良好地进行应急事故流程管理，最后回顾了同样一个事故在良好的应急事故流程管理之下的执行过程。

无流程管理的紧急事故

假设你现在是Mary，某公司的on-call工程师。现在是周五下午两点，突然间你的手机开始响个不停。黑盒监控告诉你，在某个数据中心中，你的服务突然不再处理任何用户请求了。长叹一口气后，你放下了手中的咖啡，开始着手修复服务。几分钟之后，另一条报警信息告诉你现在另外一个数据中心也出现了问题。接下来，总共5个数据中心，其中的3个都出现了问题。让整个情况变得更糟的是，现在用户流量已经超出了你剩下的数据中心的服务容量，它们开始过载了。还没等你做出任何反应，整个服务已经陷入过载停滞状态，彻底全坏了。

你呆呆地盯着日志信息一直看，好像时间都停止了。几千行错误日

志显示一个最近更新的模块发生了错误，于是你决定先回滚到上一个版本。

回滚并没有修复问题，于是你拿起电话开始呼叫这个正在进入休克状态的服务的主要开发者—Josephine。Josephine十分不愉快地接了电话，目前正是她当地时间凌晨3:30，但是她勉强同意登录到系统上看一眼。同时你的同事Sabrina和Robin也开始从他们的系统上尝试解决问题。“我们只是看看”，他们说道。

现在一个业务高管给你老板打了个电话，正在愤怒地质问为什么没有人通知他—“业务关键系统已经完全瘫痪”。同时，公司副总裁开始缠问你什么时候能够恢复服务，同时不停地问你，“为什么会这样？是什么原因导致的？”如果是平时的话，你也许还能够体谅他们的愤怒之情，但是在这种情况下，你哪里还有精力？这时候，公司副总裁开始根据他以前就不多的工程师经验，开始不断提出一堆难以拒绝的问题和评论，比如“增加缓存大小！”

没过多久，最后两个剩余的数据中心也全挂了。在你完全不知情的情况下，睡眼惺忪的Josephine 给Malcolm打了个电话。Malcom突然灵机一动，没准我们可以改改服务的CPU黏性设置（即让进程独占某个CPU）。他自信只要他改动一点点小东西，就能将剩余不多的几个服务器优化一下，问题就解决了。但是在他改完配置的几秒后，这些进程重启了，读取了新配置文件，然后彻底崩溃了。

对这次无流程管理的事故的剖析

首先，我们要注意到，在上面说的这个场景中，每个人都在尽力解决问题，起码在他们自己看来是这样。那么问题是怎么变得越来越糟的呢？在这次处理过程中，有几个常见的问题导致了整个事故的失控。

过于关注技术问题

我们倾向于按技术能力指标聘请像Mary这样的人。所以她在灾难过程中忙着不断改变系统，英勇地尝试去解决服务问题一点儿也不奇怪。由于她正在忙着执行技术操作，所以根本没有时间和精力去思考如何能

够通过其他手段缓解当前服务的问题。

沟通不畅

同样的原因，Mary根本没有时间清晰和有效地与其他人进行沟通，没有人知道他们的同事正在做什么。业务部门领导十分愤怒，最终用户正在面临服务问题，而其他可以帮忙调试和处理问题的工程师却没有被充分地利用起来。

不请自来

Malcolm 正在出于善意修改系统，但是他没有通知其他的同事—甚至Mary。严格地讲，Mary是故障排除的主要负责人，Malcolm的操作将服务状况变得更糟了。

紧急事故的流程管理要素

紧急事故流程管理的技巧和手段都是为了让这些富有热情的人能够真正帮上忙。Google的紧急事故管理系统是基于Incident Commad System[44]的，这套体系以清晰度和灵活性著称。

嵌套式职责分离

在事故处理中，让每个人清楚自己的职责是非常重要的。有点反直觉的是，明晰职责反而能够使每个人可以更独立自主地解决问题，因为他们不用怀疑和担心他们的同事都在干什么。

如果一个人目前要处理的事情太多了，该人需要向计划负责人申请更多的人力资源。他们应该将一部分任务交接给其他人，有的时候这些人应该负责在事故流程管理系统中创建更多的子事故（即用来通知公司其他相关部门等。）另外一种方式是，某个负责人可以将某个系统组件完全交给同事来处理，由该同事直接向负责人汇报情况。

以下是系统中可以分配给某个人的角色。

事故总控（incident command）

事故总控负责人掌握这次事故的概要信息。他们负责组建事故处理团队，按需求和优先级将一些任务分配给团队成员。未分配的职责仍由事故总控人负责。如果有必要的话，他们要负责协调工作，让事务处理团队可以更有效地解决问题，比如代申请访问权限、收集联系信息等。

事务处理团队（operational work）

事务处理团队负责人在与事故总控负责人充分沟通的情况下，负责指挥团队具体执行合适的事务来解决问题。事务处理团队是在一次事故中唯一能够对系统做修改的团队。

发言人（communication）

该人是本次事故处理团队的公众发言人。他的职责包括向事故处理团队和所有关心的人发送周期性通知（通常以电子邮件形式），同时可能要负责维护目前的事故文档，保证其正确性和信息的及时性。

规划负责人（planning）

规划负责人负责为事务处理团队提供支持，负责处理一些持续性工作，例如填写Bug报告记录系统，给事务处理团队订晚餐，安排职责交接记录。同时负责记录在处理过程中对系统进行的特殊操作，以便未来事故结束后能够复原。

控制中心

受到事故影响的部门或者人需要知道他们可以与事故总控负责人联系。在很多情况下，可以设立一个“作战室”（war room），将处理问题的全部成员挪到该地办公。其他团队可能更希望在自己的办公位处理问题，通过IRC或者E-mail关注事态进展。

Google发现IRC 对紧急事故处理非常有帮助。IRC系统非常可靠，同时可以为整个沟通过程提供记录，对处理过程中的细节记录非常有帮助。我们开发了一些IRC机器人，有的可以将事故处理的通信过程记录下来帮助事后总结分析，有的可以记录在事故过程中发出的所有报警。IRC同时也是一个分布全球的团队协调工作的良好媒介。

实时事故状态文档

事故总控负责人最重要的职责就是要维护一个实时事故文档。该文档可以以wiki的形式存在，但是最好能够被多人同时编辑。大部分Google团队使用Google Docs，但是Google Docs 团队使用Google Sites做这件事：利用你正要修复的服务来修复该服务恐怕不是什么好主意。

附录C提供了一篇示范性事故状态文档。这篇实时的文档可能比较混乱，但是必须得能够解决问题。使用模板来填写这个文档能容易一些，同时将最新信息发布在文档顶部也有助于提高可用性。在事后总结时还要使用这篇文档。

明确公开的职责交接

超出工作时间以后，事故总控负责人的职责能够明确、公开地进行交接是很重要的。如果你将事故总控职责交接给另外一个地区的人时，可以通过电话或一次视频会议将目前的情况交接给他。当新的事故总控负责人了解了目前事故情况时，当前事故总控负责人必须明确地声明："从现在开始由你负责事故总控，请确认。"当前事故负责人在得到明确回复之前不得离开岗位。交接结果应该宣布给其他正在处理事故的人，明确目前的事故总控负责人。

一次流程管理良好的事故

下面我们来看一下如何利用流程管理的理念，上面那起事故的处理结果是什么样的。

现在是下午两点，Mary正在喝她今天的第三杯咖啡。手机报警的刺耳声音吓了她一跳，于是赶紧放下咖啡处理问题。事故：一个数据中心停止服务用户请求了。她立即开始调查原因。不久之后，第二个报警来了，五个数据中心的第二个也出现了问题。因为问题正在迅速变得严重起来，她意识到通过事故处理流程管理框架来操作会更好。

Mary戳了一下Sabrina,"你可以当事故总控负责人吗？",Sabrina点了点头，开始记录Mary描述的目前问题情况。她将这些细节写成了一封电子邮件，发送给了预先设立好的一个邮件列表。Sabrina意识到她还没

有评估这次事故的影响范围，于是她转向Mary询问她的目前评估。“用户暂时还没有受到影响，我们现在只能抱希望不要再失去第三个数据中心。”Sabrina将这条状态信息记录在了一个实时事故状况文档里。

当第三个警报来临时，Sabrina从IRC频道中看到了这条警报信息，立即又发送了一封状态更新邮件。这个电子邮件列表可以让VP们粗略了解到事故的目前进度，免得他们还要过问细节。Sabrina同时请了一个外部通信代表开始起草一篇用户通知。她随后又咨询Mary现在是否需要联系研发团队负责人（目前是Josephine），得到Mary的首肯后，Sabrina联系了Josephine。

Josephine上线之后，发现Robin也自愿加入进来。Sabrina 提醒Robin和Josephine，他们应该优先处理任何Mary交给他们的工作，同时他们必须告知Mary他们进行的任何操作。Robin和Josephine 通过阅读实时事故状况文档，很快熟悉了目前的情况。

到下午5点时，Sabrina开始寻找接下来负责处理事故的替代人，因为她和她的同事们快要到下班时间了。她更新了事故状况文档。在5点45分时，她召开了一个简短的电话会议，让所有人都清楚目前的情况。在6点时，他们与他们的姐妹团队（另外一个办公室同团队的人）进行了职责交接。

Mary第二天早上回到公司时，发现她身处大西洋另一端的同事已经定位了具体问题，缓解了问题，同时将事故做了了结，已经开始写事后总结了。问题解决了！她冲了点咖啡，开始规划一些结构性改变，使得这类问题在未来不会再重现。

什么时候对外宣布事故

先宣布事故发生，随后找到一个简单解决方案，然后宣布事故结束，要比在问题已经持续几个小时之后才想起流程管理更好。应当针对事故设立一个明确的宣布条件。Google团队依靠下面几个宽松的标准——如果下面任何一条满足条件，这次事故应该被及时宣布。

- 是否需要引入第二个团队来帮助处理问题？
- 这次事故是否正在影响最终用户？
- 在集中分析一小时后，这个问题是否依然没有得到解决？

如果平时不经常使用，事故流程管理的可靠性萎缩得很快。所以怎么使工程师不忘记他们的流程管理技能呢？难道一定要制造更多事故吗？幸运的是，事故流程管理框架常常也适用于其他的跨时区、或者跨团队的常规运维变更实施。如果我们经常使用流程管理框架处理生产变更请求，那么在事故来临时，就可以很好地利用流程管理框架管理它。如果你的组织经常进行灾难恢复演习（你应该这样做！参见文献[Kir12]），事故流程管理应该包含在其中。Google经常针对之前发生的灾难进行角色扮演式演习，比如演习另外一个地区的团队处理过的问题，以更好地熟悉事故流程管理体系。

小结

我们发现，通过事前准备一个事故流程管理策略，并确保平稳实施，以及经常测试，我们能够降低事故的平均恢复时间（MTTR），同时减轻处理紧急事故的人的工作压力。任何对服务可靠性关注的组织团队都会从类似策略上获得帮助。

事故流程管理最佳实践

划分优先级：控制影响范围，恢复服务，同时为根源调查保存现场。

事前准备：事先和所有事故处理参与者一起准备一套流程。

信任：充分相信每个事故处理参与者，分配职责后让他们自主行动。

反思：在事故处理过程中注意自己的情绪和精神状态。如果发现自己开始惊慌失措或者感到压力难以承受，应该寻求更多的帮助。

考虑替代方案：周期性地重新审视目前的情况，重新评估目前的工作是否应该继续执行，还是需要执行其他更重要或者更紧急的事情。

练习：平时不断地使用这项流程，直到习惯成自然。

换位思考：上次你是事故总控负责人吗？下次可以换一个职责试试。鼓励每个团队成员熟悉流程中的其他角色。

第15章 事后总结：从失败中学习

作者：**John Lunney**、**Sue Lueder**

编辑：**Gary O ' Connor**

学习是避免失败的最好办法。

——Devin Carraway

作为SRE，我们负责运维大型的、复杂分布式系统。同时我们还在不断地增加新功能，和增加新系统。以我们的改动速率和部署规模，发生事故是难以避免的。在事故发生后，我们要修复根源性问题，同时将服务恢复到正常状态。如果没有一种方法从已发生的事故中学习经验，那么事故就可能循环反复地发生。如果不能解决这个问题，那么随着系统规模和复杂度的增加，事故可能成倍增加，最终导致我们没有足够的资源处理事故，从而影响最终用户。因此，事后总结是SRE的一个必要工具。

事后总结的概念已经是科技行业中众所周知的了（参见文献[All12]）。一篇事后总结是一次事故的书面记录，包括该事故造成的影响，为缓解该事故采取的措施，事故的根本原因，以及防止未来问题重现的后续任务。本章描述了决定是否需要书写事后总结的标准，一些事后总结的最佳实践，以及我们在如何培育一种良好的事后总结风气上的一些经验。

Google的事后总结哲学

书写事后总结的主要目的是为了保证该事故被记录下来，理清所有的根源性问题，同时最关键的是，确保实施有效的措施使得未来重现的几率和影响得到降低，甚至避免重现。详细的问题根源分析技巧请参看文献[Roo04]，这里不再详述。在系统质量分析领域有非常多的文章、最佳实践和工具可供选用。Google的每个团队在根源分析上都采用了不同的方法和工具。在任何一个重要事故发生后，团队必须书写一份事后总结。要注意的是，书写事后总结不是一种惩罚措施，而是整个公司的一次学习机会。但是书写事后总结的过程确实需要消耗团队的一定时间和精力，所以我们在选择上很严格。每个团队都有一些内部灵活性，但是基本的事后总结条件为：

- 用户可见的宕机时间或者服务质量降级程度达到一定标准。
- 任何类型的数据丢失。
- on-call 工程师需要人工介入的事故（包括回滚、切换用户流量等）。
- 问题解决耗时超过一定限制。
- 监控问题（预示着问题是由人工发现的，而非报警系统）。

在事故发生前定义好事后总结的标准是很重要的，这样每个参与事故处理的人都知道是否应该书写书面报告。在这些客观条件之外，任何受影响的相关部门都可以提出写一篇事后总结的要求。

在SRE的文化中，最重要的就是事后总结“对事不对人”。一篇事后

总结必须重点关注如何定位造成这次事件的根本问题，而不是指责某个人或某团队的错误或者不恰当的举动。一篇对事不对人的事后总结假设所有参与事件处理的人都是善意的，他们在自己当时拥有的信息下做了正确的举动。如果因为某些“错误的”举动就公开指责或者羞辱某个人或团队，那么人们就会自然地逃避事后总结。

对事不对人的文化起源于医疗和航空行业，在这些领域中，错误的举动可能会造成人身伤亡。这些行业认为每个“错误”都是一次学习机会，从而使系统变得更可靠。当事后总结系统性、逻辑性地讨论为什么某些团队或个人会在事故过程中获得“错误”的信息时，我们才能建立更好的预防措施，防止问题再现。我们不能“修好”某个人，但是可以通过改善系统和流程从而更好地协助他在设计和维护大型复杂系统时，做出更多“正确”的判断。

当一次事故发生时，我们不能把事后总结当成例行公事。我们的工程师将事后总结看作一个修复问题，一个使Google变得更可靠的机会。一篇“对事不对人”的事后总结不应该简单地指责或者抱怨某个团队，而应该确实提出服务如何能够获得进步。

下面是两个例子。

指责

“我们需要重写整个复杂后端系统。在过去三个季度中，它每周都在出问题。我们对一点一点修复它的问题已经烦透了！真的，如果我再收到一个报警，那我就自己重写了。”

对事不对人

“通过重写整个后端系统可能可以避免这些烦人的报警信息继续发生，目前版本的维护手册非常冗长，学习成本很高。相信通过重写，可以减少报警信息，未来的oncall工程师会感谢我们的。”

最佳实践：避免指责，提供建设性意见

对事不对人的事后总结有的时候比较难写，因为事后总结的格式清

晰地表明了触发事故的原因。从事后总结中排除指责的因素可以使人们在向上级汇报问题的时候更有自信。同时我们也不应该因为某个团队和个人经常写事后总结而对他们产生怀疑。一个充满相互指责风气的环境很容易让人将事故和问题掩盖起来，从而对整个组织酿成更大的灾难（参见文献[Boy13]）。

协作和知识共享

我们非常看重协作，书写事后总结的过程也不例外。事后总结工作流程的每一步都包括团队协作和知识共享。

我们的事后总结文档都是用Google Docs写的，使用一个公司的内部模板（参见附录D）。不管具体采用哪些工具，请确保优先选择下列功能。

1.实时协作

使得写作过程可以很快地收集数据和想法。这在事后总结早期很有帮助。

2.开放的评论系统

使大家都可以参与进来提供解决方案，以及提高对事故处理细节的覆盖程度。

3.邮件通知

可以在文档中给其他用户发消息，或者引入其他人来共同填写文档。

书写事后总结的过程还包括正式的评审和发布过程。在实践中，团队首先内部发布，同时有目的地找一些资深工程师来评估文档的完整程度。评审的条件包括如下几项。

- 关键的灾难数据是否已经被收集并保存起来了？
- 本次事故的影响评估是否完整？

- 造成事故的根源问题是否足够深入？
- 文档中记录的任务优先级是否合理，能否及时解决了根源问题？
- 这次事故处理的过程是否共享给了所有相关部门？

初期评审结束之后，该篇事后总结会在更大范围内公布。通常是在更大范围的研发部门内部传阅，或者是以内部邮件列表形式传播。我们的目标是将事后总结传播得越广越好，传递给所有能够以此获益的团队和部门。Google针对任何可以定位个人[45]的信息有严格管控规则，事后总结这种内部文档也不允许包括这样的信息。

最佳实践：所有的事后总结都需要评审

未经评审的事后总结还不如不写。为了保障每个写完的草稿都得到评审，我们鼓励定期举行评审会议。在这些会议上，我们应该注意着重解决目前文档中的疑问和评论，收集相关的想法，将文档完成。

一旦所有的事故参与者都对文档和其中的代办事项表示了肯定，这篇事后总结会被添加到该团队或者整个组织的文档汇总中。[46]透明化的共享机制保证了每个人都可以很容易地找到和学习以前的事故。

建立事后总结文化

说起来容易做起来难，建立起事后总结文化需要不断地培育和加强。Google通过高级管理层的主动参与协作和评审环节来不断加强内部事后总结文化。虽然管理层可以鼓励建立“对事不对人”的氛围，但是由工程师自主驱动，效果会更好。为了更好地建立事后总结文化，SRE经常搞一些集体学习活动，示例如下：

本月最佳事后总结

通过每周一次的新闻邮件，与整个组织共享一篇有趣并且质量很高的事后总结。

Google＋事后总结小组

本小组共享和讨论内部与外部的事后总结，同时包括一些最佳实践和事后总结的评论文章。

事后总结阅读俱乐部

某团队经常性地组织阅读俱乐部。在这项活动过程中，所有参与者和未参与者（甚至是新来的工程师）共同开发式地讨论一篇有趣或者很有影响力的事后总结，包括事件的发生过程，学习到的经验教训，以及善后处理。通常，我们讨论的是几月前甚者几年前的事后总结。

命运之轮（wheel of misfortune）

刚加入的SRE工程师经常需要参加命运之轮这个活动（具体参见第28章的“故障处理分角色演习”一小节）。在这个活动中，我们将之前的某篇事后总结的场景再现，一批工程师负责扮演这篇文档中提到的各种角色。经常，当时的事故总控负责人也参与其中，确保这次演习越真实越好。

在引入事后总结机制的时候，最大的阻力来源于对投入产出比的质疑。下面的策略可以帮助面对这些挑战：

- 逐渐引入事后总结流程。首先进行几次完整的和成功的事后总结，证明它们的价值，同时帮助确定具体书写事后总结的条件。

- 确保对有效的书面总结提供奖励和庆祝。不光通过前面提到的公开渠道，也应该在团队或个人的绩效考核中体现。

- 鼓励公司高级管理层认可和参与其中。Larry Page（Google创始人之一）经常称赞事后总结的价值之高！

最佳实践：公开奖励做正确事的人

Google创始人Larry Page 和 Sergey Brain，每周会在美国加州山景城总部举办全公司的TGIF大会，所有Google全球办公室都可以收看直播。2014年一次TGIF以“事后总结的艺术”为主题，邀请了SRE一起讨论重大的事故处理过程。某SRE讨论了他经历的一次更新事故。在某次更

新中，虽然经过了详细测试，还是由于不可预知的关联系统问题导致某关键服务停止运行4分钟。因为SRE立刻执行了回滚，从而避免了时间更长和影响更大的事故，导致这次事故仅仅持续了4分钟。这名工程师不仅立即收到了两个同事发来的奖金（Peer Bonus[47]），同时他的快速和沉着处理赢得了TGIF观众的热烈掌声，其中包括几千名在场观众和两名Google创始人。除此之外，Google内部还有一系列内部社交网络，鼓励用户对质量优秀的事后报告和灾难处理提出奖励。上面这个例子只是很多例子中的一个，在Google内部，良好的事后总结和事故处理可以赢得从CEO到工程师的一致好评。[48]

最佳实践：收集关于事后总结有效性的反馈

在Google，我们倾向于解决新问题并将创新在内部共享。我们经常在团队内部搞调查问卷，以了解事后总结流程是否在合理地支持他们的工作，以及如何改进。我们问的问题有：团队文化是否支持你的工作？书写事后总结是否引入了太多杂事（见第5章）？你们的团队有什么最佳实践可以分享？这些调查结果可以给平时很忙的SRE一个机会去提供有效性的改进和反馈意见。

在事故流程管理之外，事后总结已经成为Google文化的一部分：在任何严重问题发生时，都会产生一篇事后总结（比如某项产品上线之后反响很差，也会有类似的事后总结）。

小结以及不断优化

我们可以很自信地说，由于我们不断地培育公司事后总结文化，Google 的事故越来越少，用户体验也越来越好。我们的“事后总结”小组是建立“对事不对人”文化努力的一个代表。这个小组协调公司内部各种部门的事后总结流程：建立事故总结模板，用流程管理工具自动化数据收集，以及自动化元数据收集以便进行趋势分析。我们能够将最佳实践共享给不同的产品部门，包括YouTube、Google Fiber、Gmail、Google Cloud、AdWords及Google Maps等。这些截然不同的产品部门都是为了共同的学习目标进行事后总结的工作。

每月有大量的事后总结在Google内部形成，汇总这些总结的工具也

越来越有用。这些工具帮助我们在事后总结中寻找共同的模式和主题，以便形成改进意见。为了更好地进行数据收集和汇总分析工作，我们最近在模板中增加了一些元数据（参见附录D）。接下来我们还会采取机器学习等手段在这个领域内尝试预测系统的弱点，降低重复事故的发生，和更好地进行实时事故调查。

第16章 跟踪故障

作者：**Gabe Krabbe**

编辑：**Lisa Carey**

提高可靠性的唯一可靠的方法论是建立一个基线（baseline），同时不断跟踪改变。Google使用Outalator—一个故障跟踪工具来做这件事。Outalator系统被动收集监控系统发出的所有报警信息，同时提供标记、分组和数据分析功能。

系统性地从过去发生过的问题中学习是服务运维的必要手段。事后总结（参见第15章）为单个故障提供了详细的信息，但是它们只是整个解决方案中的一部分。只有影响非常大的故障才会进行事后总结，所以小型但是频繁发生的故障经常不会被包含在内。同样的，故障总结常常用于讨论对整个服务或者整套服务体系的改进意见，有可能错失一些对小型服务的改进意见的讨论。同时单独故障的事后总结有可能错过一些从全局看来非常有用，但是对单独故障不那么划算的改进讨论。[49]

我们还可以从这些问题中获得如下信息："每次on-call轮值发生的报警次数是多少","上个季度中的可操作的报警和不可执行的报警的比例是多少","本团队管理的服务中，哪个消耗的人工最多"。

Escalator

在Google中，SRE接收到的所有报警信息都由一个中央性的、高可用的服务管理。该服务管理某项报警是否得到了回应。如果在配置时间周期内没有收到回应，这个系统将按照升级规则（escalation path）将

报警升级给另外一个预先配置的目的地。比如从主on-call工程师升级给副on-call工程师。这套系统名为The Escalator，一开始的设计目标是透明接受on-call邮件列表的CC副本。这使得Escalator 可以非常容易地和现有的工作流整合起来，不需要改变用户习惯（在当时，不需要改变监控系统的行为。）

Outalator

在创建Escalator之后，为了更好地在现有基础设施上增加新功能，我们创建了一个处理高级抽象概念“故障”的系统——Outalator。

Outalator允许用户将多个“队列”（queue）收到的报警信息按时间顺序交叉列出，而不需要切换队列视图。图16-1显示了Outalator队列视图中的多个队列同时显示。这项功能十分有用，因为某个SRE团队经常需要同时处理多个服务的报警信息。

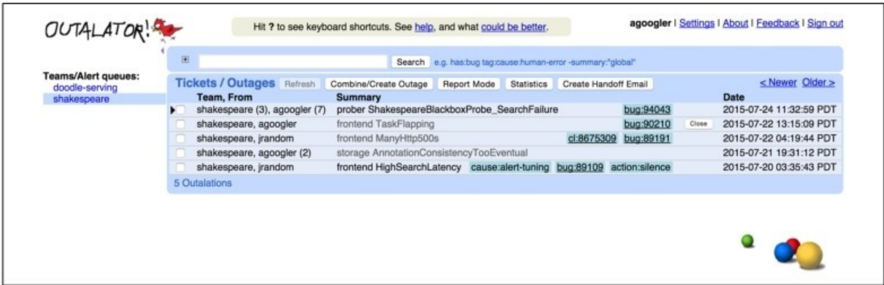


图16-1:Outalator 队列视图

Outalator 将每条原始通知都保存在数据库中，同时允许给故障增加标记。为了方便起见，Outalator同时也接收和保存处理故障过程中的邮件记录。Outalator还允许将某条回复标记为“重要的”，因为有时候，某条回复对其他人意义不大（比如某条E-mail回复只是为了将某个人加入CC列表）。默认Outalator只显示“重要的”回复，其他回复记录都会被折叠起来，以保持页面简洁。所有这些功能加起来，可以帮助我们更好地从一堆分散的邮件信息中获取有价值的信息。

在Outalator中，多个报警可以被合并成一个。这些报警通知可能是由同一个故障造成的，也可以由非相关的事件组成，或者没有价值的记

录性信息，或者只是暂时的监控失效报警。这项分组功能，如图16-2所示，可以进一步减少显示界面中的无关信息，同时将分析“每天的报警”和“每天的故障”两种场景分开。

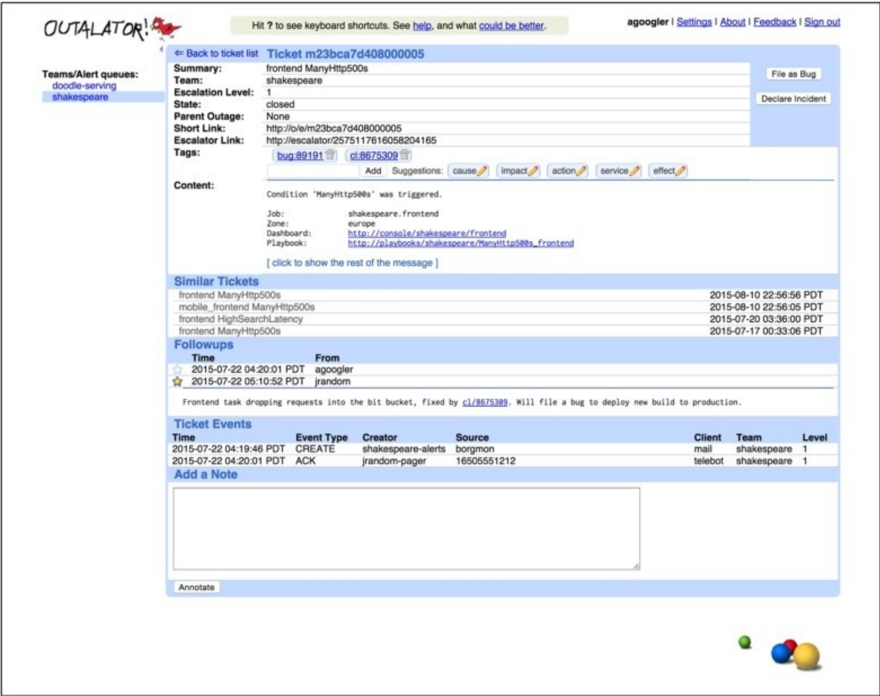


图16-2:Outalator 针对某项故障的视图

打造你自己的Outalator

很多组织都在使用即时消息系统，比如Slack、HiPchat，甚至使用IRC进行内部沟通和Dashboard更新。这些是接入一个类似Outalator系统的好地方。

聚合

某一个单独的故障，可能（经常会）触发多个报警。例如，网络问题导致所有服务都超时和无法连接后端服务器，所以每个受影响的团队都会收到自己服务的报警。同时，这也包括运维后端服务器的人。同时网络运维中心（NOC）也会收到自己的报警信息。同样的，有时候一个更小的，仅仅影响某个单独服务的故障也可能同时满足多种错误状态而同时触发多项报警。试图减少同一个故障造成的报警信息总数量是值得

做的，但是有时候在考虑过滤掉虚假报警以及漏报报警的危险性时，可能无法完全避免。

将多个报警信息“聚合”成一个单独的故障能够有效解决这个问题。有时候发送一封“这个报警和其他报警一样，是同一个故障造成的”电子邮件可能能够解决问题，但是聚合功能能更好地消除重复报警，避免重复性工作。

加标签

当然，不是每个报警信息都代表一次故障。虚假报警有时候会发生，同时也有测试产生的报警和发错的邮件信息。Outalator本身并不区分这些信息，但是提供了一种通用的标签机制，以允许对报警增加元数据信息。标签基本上没有格式要求，都是独立的“单词”。冒号（:），被认为是语义性分隔符，所以这实际上隐含提倡了一种层级结构的命名空间体制，以及一些自动化处理。Outalator使用一些建议的前缀，类似“cause”和“action”等作为命名空间，但是每个团队都可以自己定义。比如“cause:network”可能对某些团队来说已经足够了，其他的团队可能需要 “cause:network:switch”和“cause:network:cable”。有些团队经常使用 “customer:123456”这类的标签，以标记某个客户。

标签也可以被解析成一个链接（比如“bug:76543”将被解析成Bug跟踪系统的一个链接）。其他一些常见的标签都是单独的词语，例如“Bogus”常常被用来标记虚假报警。当然，有时候一些标签是拼写错误造成的（如“cause:netwrrk”），同时有些标签没那么有用（“problem-went-away”）。通过不预先提供标准列表而是允许用户自己制定标准使得整个工具更有用，数据也更有用。总体来说，标签机制对团队来说非常有用，可以在不做详细分析的情况下快速看到服务的一些问题。虽然这个功能看起来并不起眼，但它却是整个Outalator中最有用的特殊功能。

分析

SRE当然不仅仅是响应和处理故障。历史数据对响应某个故障来说也很有用。我们可以通过查找“上次我们做了什么”作为起点处理本次故

障。但是历史数据对分析系统性的、周期性的，以及更广范围内的系统问题也很有帮助。故障跟踪工具最重要的功能之一就是使用户可以进行这类分析。

底层的分析包括计数和基本的汇总统计报告。报告细节每个团队不同，但是一般包括每周/每月/每季度的故障数量和每次故障的报警数量。下一层分析更重要一些：对比团队/服务以及按时间分析趋势和模式。这一层分析允许团队使用自己的历史数据和其他团队的数据来判断某项报警是否“太吵了”。“这是本周的第三次报警了”怎么理解都可以，但是通过某项报警过去是“每天5次”，还是“每周5次”，就很有对比意义了。

数据分析的下一步是找到影响更广泛的问题，不仅仅是简单的计数，而是需要一些语义分析。举例来说，通过找到基础设施中造成故障最多的一部分，可以更好地知道如果提高该部分的稳定性或性能会带来多大帮助。[50]当然，这需要建立在故障记录中已经提供这方面信息的基础上。一个简单的例子：不同的团队有服务自己的报警，类似“过期数据”或者“延迟过高”等。这两种报警都可能由网络阻塞问题造成数据复制延后导致。或者，它们可能是符合设计的服务质量目标的（SLO），只是无法满足用户的期望而已。通过跨团队收集这些信息，可以从中找出系统性的问题，同时提供正确的解决方案。甚至有的时候，我们需要人为制造一些宕机时间，以免给内部用户造成某种假象（通常指某些服务设计的架构已经决定发生故障会耗时很久才能解决，但是经常由于运气因素而造成非常稳定的假象。某些团队选择定期制造人为宕机时间，以避免用户过于依赖该服务）。

报告和公告

Outalator 中对一线SRE更有用的功能是可以选择一系列故障，将它们的标题、标签和“重要的”记录信息用邮件格式发送给下一个on-call工程师（也可以CC其他人或邮件列表）。这样可以很容易地进行交接工作。Outalator同时支持一种“报告模式”，为周期性的生产服务评审（大部分团队每周进行一次）提供帮助。在这个模式下，所有“重要的”评论都内嵌在列表中展开，以提供更好的视图。

能够将某条报警，或者某系列报警跟某个其他故障链接起来有明显的好处：可以加快检查速度和通过确认目前的确有某项故障以降低其他团队的压力。同时也有一些不那么明显的好处。用Bigtable举例来说，如果某项服务故障由明显的Bigtable故障造成，而Bigtable貌似还没有报警产生，那么手工创建一条报警可能是个好主意。让故障信息跨团队可见，可能在问题处理过程中提供巨大帮助，至少不会帮倒忙。

有些团队甚至设立了一些虚设的Escalator 队列：这些队列并没有真人接受报警，但是这些报警仍然会出现在Outalator中，并且可以打标签、评论分析以及复审。例如，“记录系统”使用这个方法记录“高权限用户”或者“特权角色”（role-account）的使用情况（但是这里必须说，这里提供的功能仅仅是基础的、技术性的，而非法律意义上的审核使用）。另外一个例子是记录某些非幂等周期性任务的执行，比如自动利用代码库中新的版本更新数据库结构。

第17章 测试可靠性

作者：**Alex Perry**、**Max Luebbe**

编辑：**Diane Bates**

如果你还没有亲自试过某件东西，那么就假设它是坏的。

SRE的一项关键职责就是要定量地分析我们维护的某项服务的质量。SRE采用将经典的软件测试技术应用在分布式系统上[51]来做到这一点。对服务质量的自信可以用过去的系统可靠度和未来的系统可靠度来衡量。前者可以通过抓取和分析历史性监控信息来获得，后者可以用基于历史数据的预测来量化。为了让这些预测信息足够准确，必须满足下列条件中之一：

- 在这段时间内，该系统完全没有改变。包括没有任何软件更新以及服务器数量变化，这意味着未来的行为方式应该与过去的行为方式类

似。

- 你可以充分描述整个系统的所有改变，这样可以针对每个系统变化引入的不确定性进行分析。

测试，是一个用来证明变更前后系统的某些领域相等性的手段。[52]每个在变更前后都能通过的测试降低了分析系统可靠性变化中的不确定性。完善的测试可以提供足够的细节信息，以帮助我们有效地预测某个系统未来的可靠度。

测试的数量直接取决于所服务系统的可靠性要求。随着代码的测试覆盖度上升，每次改动的不确定性和降低系统可靠度的可能性都降低了。足够的代码测试覆盖度可以让我们对系统做出更多的改动，而不会使系统可靠度下降到可接受水平之下。如果我们在短时间内做了大量改动，那么预测的系统可靠性就会趋近于可接受的极限。这时，我们应该停止引入新的改动，等待监控数据再累积一段时间。累积的监控数据和测试的覆盖数据可以用来证实新的代码执行路径的可靠程度。假设该服务的客户端是随机分布的（参见文献[Woo96]），我们可以通过监控指标的采样统计数据推断出系统行为是否发生了变化。这些统计数据指出了需要完善和调整测试的区域。

测试和平均修复时间的关系

系统通过某项测试或者一系列测试并不一定能证明系统是稳定的，但是失败的测试通常证明了系统不可靠。

一个监控系统也可以发现系统中的Bug，但是仅限于汇报机制的速度（例如报警规则的间隔等）。平均修复时间（MTTR）是衡量运维团队通过回滚或者其他动作修复某个Bug的时间。

一个测试系统可以检测出一个MTTR为0的Bug。这种情况出现在当一个系统级别的测试应用在某个子系统上，并且测试检测到了监控系统也会检测到的一个Bug。那么这项测试可以被用来阻挡这个Bug的发布，使得这个Bug根本不会进入生产环境（虽然这个Bug可能还是需要从源代码级别修复）。通过阻止发布的方法修复MTTR为0的Bug是很高效的。

你发现的0 MTTR Bug越多，你服务的平均失败时间（MTBF）就越长。

随着测试的优化，MTBF会上升，开发者就可以更快地上线新功能。有些新功能当然会包含Bug。新的Bug的发现和解决会同时导致变更发布速率的下降。

有关测试的文章基本大同小异。分歧主要发生在不同措辞上，以及对不同软件生命周期阶段中对测试重要性的认知上。有关Google内部测试的资料请参看文献[Whi12]。接下来的一节主要讲述与软件测试相关的术语在本章中的应用。

软件测试的类型

软件测试基本分为两大类：传统测试和生产测试。传统测试在软件开发过程中很常见，主要用来在开发过程中离线评估软件的正确性。生产测试在生产Web服务器上进行，用来评估一个已经部署的软件系统工作是否正常。

传统测试

如图17-1所示，传统软件测试由单元测试开始（unit test）。更复杂的测试都是在单元测试之上进行的。

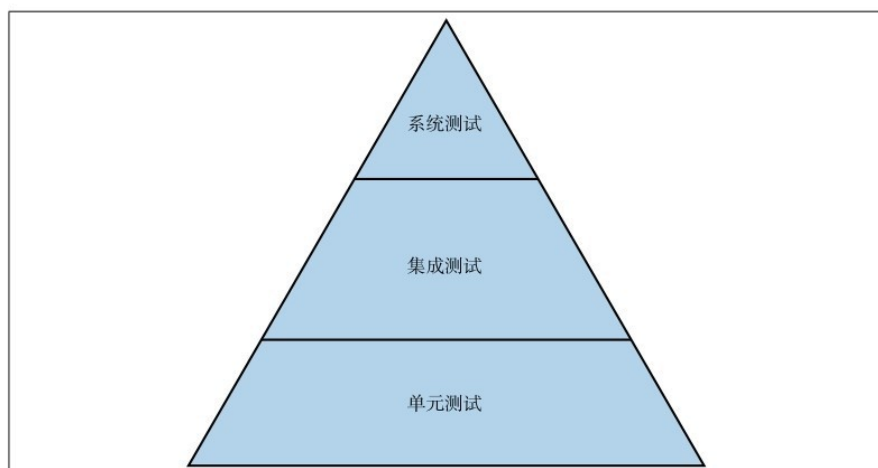


图17-1：传统测试的层级模型

单元测试

单元测试（unit test）是最小、最简单的软件测试形式。这些测试用来评估某一个独立的软件单元，比如一个类，或者一个函数的正确性。这些测试不考虑包含该软件单元的整体系统的正确性。单元测试同时也是一种规范，用来保证某个函数或者模块完全符合系统对其的行为要求。单元测试经常被用来引入测试驱动开发的概念。

集成测试

通过独立的单元测试的软件组件被组装成大的系统组件。工程师通过在这个组件中运行一个集成测试（integration test）来检验该组件的功能的正确性。依赖注入（dependency injection），利用类似Dagger[53]这样的工具，我们可以创建出复杂依赖的mock（测试中替代真实逻辑的伪组件），用以方便地测试某个系统组件。一个常见的例子是通过依赖注入来用轻便的mock替换一个有状态的数据库，同时保持一模一样的行为特征。

系统测试

系统测试（system test）是一个在未部署的系统上运行的大型测试。某个组件的所有模块（Module）都会被装载到系统中（例如通过集成测试的软件服务器）。接下来工程师可以端到端地测试系统功能。系统测试包括以下几种类型。

冒烟测试（smoke test）

工程师在冒烟测试中可检测非常简单但是非常重要的系统行为。这是最简单的一种系统测试形式。冒烟测试有时也被称为理性测试，如果该测试不通过，那么其他更昂贵的测试可以不用运行了。

性能测试（performance test）

一旦冒烟测试通过，系统基本的正确性已经得到了保障。下一步通常是通过某个系统测试的变形来保证整个系统的性能自始至终保持在可接受范围内。因为系统的响应时间和资源要求可能在开发过程中大量改

变，该系统必须接受某些测试以确保它不会在没人知道的情况下逐渐变慢（在发布到最终用户之前）。例如，一个程序可能随着改变开始需要32GB内存，而以前只需要8GB。或者该程序的响应时间由10ms变成了50ms，随后变成了100ms。性能测试可以保证随着时间推移系统性能不会下降，或者资源要求不会升高。

回归测试（regression test）

另外一种系统测试可保证之前的Bug不会重现。回归测试可以被理解为曾经发生过的，导致系统故障或产生错误信息的Bug列表。通过将这些Bug记录为系统测试或者集成测试，重构代码的工程师可以保证他们不会偶然间将他们曾经辛苦调查和修复的Bug又带回来。

很重要的是，每个测试都有成本，时间成本和计算资源成本。在一个极限上，单元测试非常便宜，通常可以在毫秒级和很少的资源上（例如一个笔记本电脑上）完成。而在另一个极限上，将一个完整的软件服务器设立起来，同时包括它所有的依赖系统（或者是mock），然后运行相关的测试可能会需要很长时间—几分钟到几小时—一般还需要专属的运算资源。时刻关注这些测试的成本，是软件开发效率提升的重要因素，同时也鼓励程序员更有效地利用我们的测试资源。

生产测试

生产测试和一个已经部署在生产环境中的业务系统直接交互，而不是运行在密闭的测试环境中。这些测试和黑盒监控在很多地方十分类似（参见第6章），有的时候也被称为黑盒测试。生产测试对运行一个可靠的生产环境来说是必要的。

变更发布与测试

人们经常宣称测试是在（或者应该在）密闭的测试环境下进行的（参见文献[Nar12]），这句话暗示了生产环境是非密闭的。当然生产环境通常不是密闭的，因为有节奏的变更发布过程在进行小范围的、已知的生产系统改变。

为了更好地控制不可预知性，同时避免最终用户受到影响，变更可

能没有按照它们被加入源代码版本控制系统的顺序发布。发布过程经常按阶段进行，使用某种机制逐渐将用户切换到新系统上。同时，监控每个新阶段以保证新环境没有遇到任何未知的问题。所以，整个生产环境通常并不能用任何一个源代码版本控制系统的版本来代表。

有可能在源代码版本控制系统中同时存在多个版本的二进制文件，以及它对应的待发布配置文件。这个场景下针对生产环境测试有可能会导致问题。例如，这项测试可能在使用代码仓库中的最新版本的某个配置文件与生产系统中的老版本的二进制文件。或者测试可能使用的是旧版文件，结果发现了一个已经被新版本修复的Bug。

同样的，系统测试也可以使用配置文件在测试前组装模块。如果测试通过，但是该测试的配置测试（下一节提到）却没有通过，那么这项测试只通过了密闭环境下的测试，却没有在生产环境下通过。这样的结果是很不好的。

配置测试

在Google内部，Web服务的配置信息保存在代码仓库中的文件中。针对每个配置文件，有一个单独的配置测试检查生产系统，以确保该服务的配置和配置文件一致，同时汇报任何不一致的情况。这个测试自然不是密闭性的，因为它们需要在测试系统的沙盒之外运行。

配置测试是针对某一个已经提交的配置文件构建和运行的。比较目前测试通过的版本和自动化系统的目标版本可以暗示我们生产环境目前与实际工作相差多远。

这些非密闭性的配置测试作为分布式监控系统的一部分是尤其有价值的，因为这些测试的在生产环境中的通过/失败状况可以揭示整个系统中配置组合错误的部分。监控系统尝试将实际用户的请求处理路径（从跟踪日志中获取）与一系列黑名单进行对比。如果发现有匹配的，则会产生报警以便停止目前的变更发布过程，并采取修复措施。

当生产环境使用配置文件的实际文件内容，同时提供一个实时查询接口时，进行配置测试是非常简单的。在这种情况下，测试代码发送一

个查询，同时将返回结果与配置文件进行对比。当配置代码做以下事情的时候，测试就变得复杂起来：

- 隐式地加入二进制文件中包含的默认值（这样查询返回的结果就和提交的版本不一致）。

- 配置文件作为一个预处理器（例如Bash）的命令行参数使用（导致结果取决于预处理器的结果）。

- 根据共享的运行时的某个特殊行为（使得测试还要取决于该运行时的版本）。

压力测试

为了安全地操作某个系统，SRE需要理解系统和组件的性能边界。在很多案例中，单独的组件在超过某个临界值时并不能优雅地降级，而是灾难性地失败。工程师使用压力测试来找到Web服务的性能边界。压力测试能够回答以下问题：

- 数据库容量满到什么程度，才会导致写请求失败。

- 向某应用每秒发送多少个请求，将导致应用过载并导致请求处理失败。

金丝雀测试

金丝雀测试（Canary test）没有包含在上面的生产测试列表中。“金丝雀”一词来源于“煤矿中的金丝雀”这样一个说法，指代利用一只鸟来检测有毒气体以避免人类中毒的做法。

要进行一次金丝雀测试，一小部分服务器被升级到一个新版本或者新配置，随后保持一定的孵化期。如果没有任何未预料的问题发生，发布流程会继续，其他的软件服务器也会被逐渐升级到新版本。[54]如果发生了问题，这个单独修改过的软件服务器可以很快被还原到已知的正常状态下。我们经常用“烘烤这个二进制文件（baking the binary）”来指代这个过程。

金丝雀测试并不真的是一个测试，而是一种结构化的最终用户验收测试。配置测试和压力测试可以测试在某种特定情况下的服务的特殊表现，而金丝雀测试更多的是比较随意的。这种测试将代码置于比较难以预测的生产环境的实时用户流量之下，看代码是否产生问题。因此，这种测试是不完美的，有的时候会漏掉某些Bug。

为了提供一个现实的金丝雀测试案例，我们来考虑在一个指数型升级部署流程（先部属1台，然后部属2台，然后4台等）下，一个对用户流量影响不大的Bug情况。我们预计，总的累积公式为CU＝RK（R是错误发生的速率，U是错误等级，见后文，K是用户流量增长172%（呈自然对数 e倍率增长）所需的时间。[55]

为了避免对用户造成影响，部署流程一旦遇到不可接受的问题需要快速回滚到之前的配置。在部署开始后到自动化系统收集到相应的异常信号并且做出回滚响应之前的一段时间内，有可能产生了一些额外的错误报告。所有这些错误报告可以用之前的累计值C和速率R代表。

将这些值除以K可以得出一个估计值U，也就是错误等级[56]。

● U＝1：用户请求只是触发了某段逻辑错误的代码。

● U＝2：用户的请求可能会随机损坏到未来该用户可能会访问的数据（意味着该问题必须严格按顺序访问才能重现）。

● U＝3：随机损坏的数据也包括前序请求需要使用的数据（意味着该问题不可重现）。

大部分的Bug都是U＝1级别的：这些Bug的出现频率与用户流量成线性正比（参见文献[Per07]）。我们一般可以通过查看日志将一些异常的请求回应变成新的回归测试来避免它们未来重现。但是对高阶Bug来说，这样的策略就没用了：在高阶Bug作用下，在所有前序请求按顺序执行过之后某一个不停失败的请求，可能在打乱前续请求的环境里不再失败了。在发布过程中，能够找到这种Bug非常重要，否则，我们的运维压力会增长得非常快。

当采取指数型发布策略的时候，一定要将前述的低阶Bug和高阶Bug

的区别记在心中。有的时候发布流程不需要绝对公平地区分用户流量，只要这种区分方式能够在同样的K时间内增长同样的倍数，U的估计值就将是有效的。哪怕你不知道哪种区分方式真正能够触发这个Bug。顺序地使用多个流量区分方式可以提供一些冗余，使K减小。这种策略可以有效降低最终用户遇到的问题累计值C，同时仍然可以尽早提供一个U的预估值（这当然最好是1了）。

创造一个构建和测试环境

虽然在项目开工第一天就开始考虑测试的类型以及各种失败场景是很美好的愿望，实际上SRE经常在一个项目已经开工一段时间之后才加入团队。这时，这个团队的项目可能刚刚验证了他们的研究模型，也可能刚刚验证了他们的算法是可以扩展到更大规模的，甚至可能这时仅仅是所有的UI　Mock刚刚被批准。这时的项目代码仅仅停留在原型阶段，完整的测试还没有写出来，甚至没有设计出来。在这种情况下，从哪里开始测试呢？如果目前基本没有测试，那么针对每个关键函数和类编写单元测试可能看起来是一项非常困难的工作。然而，我们可以将测试的重点集中在用最小力气得到最大收益的地方。

可以先从如下的问题开始：

- 是否能够将源代码按重要程度分出优先级？用研发管理和项目管理的行话来说，如果所有任务都是高优先级的，那么它们就也全是低优先级的。是否可以将要测试的系统组件按重要度排序（用什么标准来衡量重要度都可以，关键要排序）？

- 是否有某些函数或者类是非常关键的，或者对业务运营极为重要的？举例来说，涉及计费系统的代码通常对业务来说是很关键的。同时计费代码也经常可以从其他部分很干净地剥离开进行测试。

- 哪些API是其他团队需要集成使用的？有时最终用户级别的测试能够找到的问题比想象的要严重，因为它们可能会迷惑其他开发团队，使他们写出错误的（或者只是效率低）API客户端代码。

发布明显的故障代码是开发者最备受指责的行为。在发布前建立一

系列冒烟测试可以抓住大部分的明显问题。这种低成本、高收益的行为应该作为第一步，从而建立起更可靠、更全面测试过的软件。

一种建立强测试文化的方式[57]是将所有遇到的问题都进行测试案例化。如果每个Bug都变成了一个测试用例，每个测试用例都应该在问题修复好之前处于失败状态。随着工程师修复好Bug，测试用例也一个一个通过，这样用不了多久你就会有一套完善的回归测试体系了。

建立良好测试的软件的另外一项关键任务是建立起一套良好的测试基础设施。测试基础设施的基础是一套版本化源代码控制体系，可以追踪源代码的每一个改变。

一旦这套体系建立起来之后，我们就可以增加一套持续构建系统，将每一次代码改变都进行一次构建。我们发现最有效的方式就是，构建系统可以实时通知提交这个改变的工程师，随时保持最新版本的代码处于正常工作状态非常重要。当构建系统通知搞坏代码的工程师时，他们应该放下手中的一切其他任务，优先处理该问题。这样做的原因在于：

- 如果问题引入系统之后，又有新的变动，修复会更难。
- 不工作的代码会对团队造成影响，因为它们必须手动绕过这些问题。
- 定时地每晚构建和每周构建将失去意义。
- 团队响应紧急发布的能力将会受到严重影响，甚者非常复杂和困难（例如，发现代码中以及依赖中的安全漏洞等）。

稳定性和敏捷性通常在SRE的世界中是互相矛盾的。但是上面最后一点反而证明了有时候敏捷性需要靠稳定性来驱动。如果每次代码构建的结果都坚如磐石非常可靠时，开发者才能迭代得更快！

某些构建系统，如Bazel[58]，为精确地控制测试执行提供了非常有价值的功能。例如，Bazel可以为软件项目生成依赖图。当某个文件改动后，Bazel可以仅仅重新构建项目中受影响的部分。这样的一套系统可以提供可重现的构建结果。这样不必每次重复运行所有测试，只运行修改

部分。于是，测试成本变得更低，也执行得更快。

有很多工具可以帮助量化和可视化测试覆盖程度（参见文献[Cra10]）。使用这些工具可更好地聚焦你的测试：将创建一个高度测试过的代码库从一个复杂的哲学理念变成一个工程项目。与其不停地重复说：“我们还需要更多的测试”，不如设立更准确的目标和期限。

要注意，不是所有的软件项目都可以平等对待。某些人身安全相关或者业务收入相关的系统相对一些非生产使用的脚本来说当然需要对应更高的测试质量和测试覆盖度。

大规模测试

我们已经描述了测试的基础知识，现在来看一下SRE是如何系统化地在大规模系统上进行测试以提高稳定性的。

小型的单元测试可能只有很少的依赖：一个代码文件，测试类库，运行时库，编译器和本地硬件。一个可靠的测试环境要求这些测试分别也有它们自己的对应测试覆盖。这些测试应该有针对性地覆盖整个测试环境中组件相互作用的部分。如果某个单元测试依赖了运行时中某个没有测试覆盖的代码，那么环境中一个毫不相干的改动[59]可能会导致该测试执行结果永远是通过，失去了测试的意义。

相反，某项发布测试可能依赖特别多，以至于它最终间接地依赖了整个代码仓库中的所有代码。如果一项测试需要一个绝对“干净”的生产环境才能进行，那么每个小改动都需要一套完整的灾难复原周期才能进行。实际的测试环境常常通过选择某几个版本分支点（branch point）进行合并，以使用最少次数的迭代周期解决最大程度的不可预知性。当然，如果某次测试失败，常常意味着你需要再选择更多的分支点，以尽快确定出错的版本。

测试大规模使用的工具

SRE的工具也是软件项目，也需要测试[60]。SRE开发的工具可能负责以下操作：

- 从数据库中获取并且传递的性能度量指标。
- 用度量指标预测未来用量，进行容量规划。
- 重构某个用户不可见的备份副本中的数据。
- 修改某些文件。

SRE工具具有两个特点：

- 这些工具的副作用基本处于被良好地测试过的主流API范围内。
- 由于现存的验证和发布流程，这些工具基本不会对用户造成直接影响，

针对危险性高的软件设立防护边界

绕过常见的、大量测试过的API而进行某种操作（哪怕是基于良好理由的前提下）的软件可能会在生产系统上造成严重后果。举例来说，某数据库引擎为了便于管理者缩短维护窗口期，而提供了一种暂时关闭事务的功能。这个引擎可能被用来进行批量操作，但是这个引擎如果一旦意外地在用户可见的实例上运行，可能会造成用户可见的数据问题。应该利用设计方式避免这种大问题：

1.使用一个独立的工具在复制配置文件中设立一道防护边界，确保该副本无法通过健康检查。于是这个副本永远不会被负载均衡选择直接面向用户。

2.修改危险的工具类代码，使得它们上线之后就检查防护边界。只允许这些工具类代码访问处于不健康状态下的副本。

3.使用与黑盒监控一样的副本健康检查机制。

自动化工具也是软件项目。因为它们代表的危险性对另一层服务来说是不可预知的，针对这些工具的测试也更为隐晦。例如下列这些自动化工具：

- 数据库索引的选择。

- 数据中心之间的负载均衡器。

- 快速重排中继日志，以尽快重建主记录的工具。

自动化工具具有两个共同特点：

- 它们的实际操作都是通过调用一个可靠的、经过良好测试的API。

- 对另外一个API用户来说，它们的调用结果是不可预知或不可见的。

测试可以保证在测试前后，服务的其他部分保持正确的行为。有时候甚至可以测试通过API暴露的服务内部状态是否一致。例如，数据库可以在索引不存在的情况下，仍然提供正确结果。但是有一些API的行为在测试前后会改变（比如说DNS 缓存在执行前后TTL已经过期）。举例来说，如果一个机器上的 runlevel发生变化，导致本地DNS服务器被替换成一个缓存代理DNS服务器。虽然两个服务器都可以将DNS查询缓存一定时间，但是内部缓存的状态不太可能同时传递过去。

既然这些自动化工具需要针对环境进行变化，就需要为其他代码文件写更多的部署测试。我们如何来定义自动化工具所运行的环境呢？毕竟，如果某个自动化工具负责将容器重新调派以提高利用率，那么在某一点上该工具也要将自己所在的容器移动。如果该工具的一个新版本带来的新算法修改内存的速度过快，最后导致网络带宽无法保持镜像从而迁移失败，这就太丢人了。就算针对这种情况编写了集成测试，该测试也不太可能使用一个跟生产环境类似的模型。同时也非常不可能使用宝贵的跨大陆带宽来进行这种测试。

更有意思的是，我们的自动化工具可能正在修改另一个自动化工具依赖的环境，或者这两个工具正在同时修改对方依赖的环境！例如，某个集群升级工具可能在推进更新的时候需要消耗集群资源。同时，某个容器平衡器将会意识到这个问题，试图将该工具迁移到其他机器上。有的时候当该集群升级工具试图升级容器平衡工具的时候，循环依赖就产生了。这种循环依赖可能并不是问题，只要API提供良好的重启机制，

同时只要有人记得针对这种情况写测试就行了。

针对灾难的测试

很多灾难恢复工具都被精心设计为离线运行。这样的工具主要做以下事情：

- 计算出一个可记录状态（checkpoint state），一个等同于服务完全停止的状态。
- 将该可记录状态推送给一个非灾难验证工具，以验证状态。
- 支持常见的发布安全边界检查工具，确保启动结果是干净的。

在很多情况中，我们可以将这些功能以一种易于测试的方式编写。如果任何一个条件（离线、可记录、可加载、安全边界检查、干净启动）无法被满足，这些工具在灾难来临前就不大可能正常工作。

那些在主流API之外工作的在线修复工具的测试就更有意思了。在一个分布式系统中，你将会遇到的一大挑战就是某个正常行为可能是最终一致的（eventual consistent），这会对灾难修复过程造成灾难性的后果。举例来说，假设你正在使用离线工具调查某项数据竞争问题。离线工具一般预期得到立即一致的结果，而不是最终一致的结果，因为立即一致更加便于测试。由于修复工具和生产工具常常是分开编译的，这种情况会变得更复杂。为了解决这个问题，你可能需要在这些测试中编译一套统一的工具，以便正确地观察要测试的事务。

利用统计学工具

统计学手段，例如用于模糊测试的Lemon（参见文献[Ana07]），用于测试分布式状态的Chaos Monkey[61]以及Jepsen[62]，不一定是可以重复执行的测试。仅仅在代码修改后运行一次这些测试，不能确定地证明相应的问题已经修复。[63]但是这些手段还是很有用的：

- 这些测试可以在某次测试中，提供所有随机选择的动作执行顺序的记录。有时候仅仅是靠记录随机数发生器的种子值就够了。

- 如果该记录可以形成一个发布测试，在开始研究错误报告之前多运行几次可能很有帮助，该问题重现的频率可以帮助预估问题是否已经被修复。（如果一个问题可以100%重现，那么修复之后就是100%不可重现。）

- 利用该问题的几种不同表现形式可以帮你定位代码中可疑的区域。

- 有时候多运行几次会显示这个失败问题比想象的要严重，你可以有针对性地升级这个Bug的优先级和严重程度。

对速度的渴求

代码库中的每一个测试对代码的每一次改动都提供了对应的测试通过和不通过的信号。有时候这个信号在看起来一样的重复运行下会发生改变。我们可以根据历史记录估算出该测试实际的通过率，同时可以计算出统计学意义上的不确定性。然而，针对每个测试的每次变动都进行这项计算是不可行的。

所以，我们必须针对某些感兴趣的可能情景建立起某种假设，然后针对这些情况重复运行一定数量的测试，以获取一个足够可靠的推断。这些情景有的是无关痛痒的，但是有的是比较有用的。这些情景在不同程度上对所有测试都有影响，因为它们常常是强耦合的，所以可靠且快速得到一系列可操作的假设（哪些组件真的坏掉了）需要针对所有场景同时进行估算。

使用测试框架基础设施的工程师想要知道他们的代码——一般是一次测试所运行的一小部分代码——是否有问题。通常情况下，代码如果没问题，那么这次失败的测试可以归罪于其他人的代码。换句话来讲，工程师关心的是他写的代码是否有预料之外的数据竞争问题，从而导致这项测试更为不可靠（或者是由于其他因素不可靠的测试变得更不可靠了）。

测试截止时间

大部分测试都是很简单的，意味着它们是用一个密闭型、自包含的

测试文件在某个计算容器中运行几秒完成的。这些测试可以在工程师切换工作环境之前提供及时的反馈信息。

需要同时协调很多服务器与容器的测试通常仅仅启动就要数秒，这样的测试无法提供及时的反馈，它们只能被称为批量测试。这些测试不能告诉工程师“先不要关掉这个代码编辑器Tab”，而只能告诉代码评审者“这段代码还没有通过测试”。

一个测试的非官方运行截止时间就是当工程师转而处理下一个事情的时候。测试结果只有在工程师走神之前才有用，否则工程师下次走神可能就像XKCD 说的那样了。[64]

假设某工程师正在一个有21,000个简单测试的服务上修改代码，提交了一份修改。为了测试该项修改，我们需要对比提交前后代码库中这些测试的通过/失败结果。如果比较结果良好，那么这次修改就可以被称为可发布的。这样的比较过程可以进而鼓励工程师运行更多的发布测试和集成测试，更多的分布式测试检查系统的扩展性（检验这项修改是否需要更多的本地资源），以及复杂度（检验这项修改在某个地方使用了一个非线性的算法）。

错误计算环境因素导致错误地认为工程师的修改是有问题的几率是多少？如果每10个改动中有1个被拒绝，那么用户肯定会非常不满意。但是如果100个修改中只有一个被拒绝，用户可能比较容易接受。

这就意味着我们的准确率需要达到0.99（正确率）的42,000次平方根。（21,000项测试，提交前后各1次）。这样的话：

$$0.99^{\frac{1}{2\times 21000}}$$

预示着这些独立的测试必须要以 99.9999% 的概率正确执行（这很难）。

发布到生产环境

生产环境的配置信息通常来说保存在版本控制系统中，配置文件通常和源代码文件分开存放。同样的，软件测试基础设施通常不会使用生产环境配置文件。就算两个配置文件存放在同一个代码仓库中，配置文件的修改也常常在分支上进行，或者是在不同的目录结构下进行。测试自动化系统经常忽略这类地方。

在传统的企业环境中，开发者负责编译可执行文件，然后由另外一批管理者更新服务器。两套测试环境在最好的情况下也是很烦人的，最差情况下会严重导致可靠性和敏捷性问题。这样的分离同时容易造成重复开发工具。在Ops环境中，这种不一致可能会导致两套工具行为上的不一致。同时，这样的分离也会导致项目发布速度的下降，因为两套版本控制系统之间经常需要互相协调。

在SRE模型下，生产环境和测试环境的分离可能也会导致生产环境和测试环境的不一致，这可能会影响想要在开发环境中重现这种不一致问题的工程师。但是至少这种分离不会导致整个研发的停滞，因为危险总是不可能完全消除的。

我们来考虑一种统一的版本化和测试化场景，以便可以使用SRE方法论。迁移到分布式测试结构下最坏的可能是什么呢？正如前文所说，我们已经假设一个软件工程师能够接受一个测试系统给出1/10不正确的反馈，那么完全迁移之后我们到底能够接受什么程度的不确定性呢？很明显，某些测试覆盖区域要比其他区域更敏感。通用一点来说：某些测试的失败比其他测试预示了更大的风险。

允许测试失败

不久以前，软件项目还是每年发布一次。二进制文件由一套编译系统执行数小时，甚至数天产生，同时大部分的测试都是由人针对书面流程手工执行的。这种发布流程效率不高，但是并没有什么需要去进行自动化的。发布成本主要集中于文档化、数据迁移、用户培训和其他因素上。这些发布流程的平均失败时间（MTBF）是一年，不管再写多少测试也是一样。由于每次发布中产生了非常多的修改，肯定会有某些用户可见的问题隐藏其中。于是，上次更新的可靠性数据对下次来说没有任何意义。

有效的API和ABI管理工具，以及大规模的解释性语言让每几分钟编译和执行一次新的软件版本成为了可能。理论上来说，足够多的人[65]可以利用前述的方法完整地测试每一个版本，从而获得同样的发布质量。虽然同样的测试最终会运行在同样的代码上，用这种方式发布的版本会比年度发布的代码质量高。因为在年度测试之外，所有中间版本也得到了测试。使用这些中间版本，我们可以很自信地将发现的问题与根源问题相对应，从而确保根源问题而不仅仅是表面现象得到了修复。缩短反馈周期这种理念与自动化测试相结合是非常有效的。

如果我们让用户在一年中测试更多的版本，平均失败周期（MTBF）反而会下降，因为用户有更多的机会接触到有问题的版本。然而，我们可以从此得出哪些区域需要更多地进行测试。如果这些测试被建立起来，每项测试都可以确保问题未来不再重现。仔细的可靠性管理可以将对不确定性的限制（由代码测试覆盖度决定）与用户可见问题的控制结合起来，调节发布的节奏。这种组合可以最大化从运维中和实际用户那里得到的反馈信息。这些反馈信息可以更好地驱动测试覆盖度，从而驱动产品发布速度。

如果SRE修改了某个配置文件，或者调节了某种自动化工具的策略（而不是新实现了某个功能），道理是一样的。如果我们根据可靠程度来决定发布的节奏，那么很多时候需要将可靠性预算按功能划分，甚至（更容易的）按团队划分。在这种情况下，一个新功能研发团队的目标是控制不确定性以达到他们预期的发布节奏。SRE团队则拥有另外一套可靠性预算，以控制他们修改生产环境的节奏。

为了让服务更可靠，避免SRE团队人数线性增长，生产环境必须做到基本上无人值守运行。为了达到无人值守目标，生产环境必须能够应对小型的失败问题。当一项重大问题需要SRE手工参与时，SRE所使用的工具必须是经过合理测试的。否则，这项操作将会降低以历史数据预测未来的自信（因为可能产生了尚未发现的Bug），这种自信的降低需要等待监控系统提供更多的分析数据来排除不确定性。在本章前面“测试大规模使用的工具”一小节中提到了SRE工具如何提高测试覆盖度，这里我们讨论的是测试决定SRE可以安全使用这些工具的频率。

配置文件的存在是因为修改配置要比重新构建一个工具要快。这种快速迭代是为了保障平均修复时间（MTTR）保持在较低的水平。然而，很多配置文件的修改并不需要这种快速迭代，从可靠性的视角来看：

- 如果一个配置文件的存在是为了保证MTTR足够低，但是仅仅在系统出现问题的时候才会修改，那么这个配置文件的修改频率比系统的MTBF还要低。那么这项修改其实引入了很大程度的不确定性，因为不知道这项修改是否会对其他部分造成影响。

- 配置文件的修改比软件更新要更频繁（例如，保存了某种更新信息的配置文件），这类修改如果不认真对待也会产生重大风险。如果对这个配置文件的测试和监控没有比应用程序的做得更好，那么这个配置文件可能会对可靠性带来很大的负面影响。

正确处理配置文件的一种方式是规定每个配置文件都是上述两种可能性之一，而不是全部，同时想办法强制执行这条规定。如果你采用上述第二种做法，那么请保证：

- 每个配置文件都有足够的测试覆盖度，以确保可以经常修改。

- 在发布之前，对文件的修改需要等待发布测试的完成。

- 提供一种应急机制以确保可以在测试完成之前将文件发布。由于应急机制会影响可靠性，通常情况下可以让执行应急机制（例如）自动提交一份Bug报告，以便下次使用更好的方案处理该问题。

应急机制和测试

我们可以部属一套应急机制，以绕开发布测试，但是这样做会导致使用这个机制发布的人无法获知任何错误反馈，直到监控系统汇报最终用户已经受到影响。最好我们能在后台继续运行测试，将发布动作与正在执行的测试关联起来，然后在测试失败之后（最快速度下）标记之前的发布事件。这样，一次有问题的手动发布可以很快地被另一次（希望是没那么多问题的）手动发布而替代。理想情况下，应急机制可以抢占常规测试的工作负载，以便更快地提供反馈。

集成

用单元测试测试某个配置文件可以提高可靠性之外，针对配置文件进行集成测试也很重要。配置文件的内容（以测试角度来看）有可能对解释执行这段文件的解释器来说是灾难性的。解释性语言（Python 等）经常被用来处理配置文件，因为它们提供了一个可嵌入的运行时，同时一些简单的沙盒机制可以应对一些非恶意的代码错误。

使用解释性语言编写配置文件是有风险的，因为这种方式经常会导致无法完全定位的潜在问题。因为加载一个配置文件实际上是执行了一段程序，该程序的执行时间没有任何实际意义的上限（你不知道加载配置文件的过程需要多少资源、时间等）。在其他类型的测试之外，我们应该将这种类型的集成测试中加入小心的截止时间检查，以确保某些测试可以在合理时间内结束。

如果配置文件是用一种特殊语法的文本格式写成的，那么每种测试覆盖度都需要从零实现。使用现成的语法（如YAML）和一个经过大量测试的分析器（例如Python的safe_load）可以将维护配置文件的成本降低。精心挑选的语法和分析器可以为配置文件加载过程提供运行时上限。但是，实现者仍需要处理配置文件结构错误，最简单的处理方式经常是没有运行时上限的。更糟的是，这些策略通常都不是可靠地测试过的。

使用Protocol Buffer[66]这种架构的好处在于提前定义了格式，在加载时可以自动检查，从而避免发生更多的问题，同时能提供良好的运行时上限。

SRE的职责经常包括编写系统工程工具[67]（如果没有其他人写的话），和为服务增加可靠的验证性测试。所有工具都可能由于测试时没有发现的问题而行为失常，所以我们要在多个层面上建立防范机制。当某一个工具行为出现异常时，工程师必须对他们的绝大部分其他工具具有足够的信心，以便利用其他工具解决或者缓解之前异常工具带来的问题。保障网站可靠性的关键因素在于找到某种可预期的异常情况，然后确保有某些测试（或者某些其他工具的输入检查器）可以汇报这些问题。出现异常的工具可能并不能自我修复，甚至不能停止目前的行为，

但是最少可以在造成严重问题前汇报检测到的问题。

举例来说，假设一个配置文件包含了一系列用户（例如非联网环境下的UNIX机器内的/etc/passwd），同时针对这个文件的一项编辑意外导致了解释器处理了一半文件之后就停止了。因为最近加入的用户没有被处理，这个机器可能还在正常运行，很多用户都意识不到有问题发生。但是负责维护用户目录的工具可以很容易地发现实际存在的目录数量和（一半的）用户列表之间的区别，同时紧急汇报这种不一致情况。这个工具的作用只在于汇报问题，所以这个工具不应该试着自我修复这个错误（通过删除用户数据）。

生产环境探针

由于测试机制是通过提供确定的数据检验系统行为是否可接受，而监控机制则是在未知数据输入下确认系统行为可以接受。看起来通过测试和监控，我们可以覆盖主要的已知和未知问题，但是实际的危险性是更复杂的情况。

已知的正确请求应该成功，而已知的错误请求应该失败。将这两种情形作为集成测试通常是一个好主意。我们可以在每次发布测试时重放这些请求。将已知的正确请求进一步划分为可针对生产环境重放，和不可针对生产环境重放。这就产生了如下三类请求：

- 已知的错误请求。
- 已知的正确请求，可以针对生产重放。
- 已知的正确请求，不可以针对生产重放。

我们可以将任两类作为集成测试和发布测试，大部分测试也可以用来作为监控探针。可能从理念上来看起来，部属这种监控探针是没有意义的，因为这些测试请求已经至少被试过两次了。但是这两种请求可能由于以下几种原因不同：

- 发布测试可能将集成服务器包在一个前端服务器和一个虚假的后端服务器之间。

- 探针测试可能将集成服务器包在负载均衡的多个前端服务器和一个真实的生产后端服务器之间。

- 前端服务器和后端服务器通常有独立的发布周期，这些发布周期不会同步。

因此，生产环境监控探针测试了一种之前没有测试过的配置。

这些探针应该永远不会失败，但是如果失败了意味着什么呢？前端API（负载均衡器）或者后端API（持久化存储）在测试环境和发布环境中是不一致的。除非你已经知道这两种环境为什么存在不一致，服务很可能没有正常工作。

生产系统更新工具逐渐替换应用服务器的同时也逐渐替换掉这些探针，所以新旧探针发往新旧服务器的4种情况在持续进行着。如果生产系统更新工具检测到错误，就会回滚到之前正确的配置上。通常情况下，更新工具预计每个新开启的应用程序实例会有一段短暂的不健康状态。如果初始化没有成功，更新工具会安全地停止更新，没有用户请求会发往该新版本。本次更新保持停止状态，直到工程师有时间和精力来解决为什么出错，并且让生产环境更新工具执行回滚。

生产环境探针测试确实对服务提供了保障，提供了一种清晰的反馈信号。这种反馈信息返回得越早越有用。最好这种测试是自动化的，这样这种方式也变得可扩展。

假设每个组件都有旧版本和新版本同时运行。新版本可能会跟旧版本的服务联络，强迫新版本程序使用过时的API。或者，旧版本程序在联络新版本程序时，使用当时（在旧版发布时）还不能正常工作的API。最好能确保这些探针都有向前兼容性，同时能覆盖大部分API。

伪后端版本

当实现发布测试时，伪后端（fake backend）经常由伙伴研发团队维护，用作构建依赖。这种密闭型的由测试基础设施执行的测试可保证永远使用相同代码版本测试前端服务器和伪后端服务器。

这种构建时依赖可以提供一个可以运行的二进制文件，理想情况下，维护它的研发团队可以同时发布伪后端服务器、生产后端服务器和探针程序。同时可以将这个二进制文件加入前端服务器发布包中。

我们的监控体系应该知道每个接口的两端的所有发布版本，这种结构可以保证我们可以获取不同版本的所有组合，以确保测试还能成功通过。监控不需要一直进行，我们只需要每次发布新版本的时候进行。这样的测试问题不一定需要阻拦新的发布。

在另一方面，自动发布机制应该在理想情况下自动阻止更新，直到这种问题组合不再可能出现。另一方面，对应的研发团队也可以考虑暂时禁止某些问题实例接受请求，直到版本问题解决。

小结

测试是工程师提高可靠性投入回报比最高的一种手段。测试并不是一种只执行一次或两次的活动，而是持久不断的。写出优质的测试需要付出的成本是很大的，建立和维护测试基础框架以推行强测试文化也是一样。在未充分理解一个问题之前，我们没法修复它，而在工程领域，了解一个问题的方法只有实际度量。本章提到的方法论和工具可以为更好地度量失败与不确定性提供一个坚实的基础，可以帮助工程师在编写和发布软件时，推演可靠性。

第18章 SRE部门中的软件工程实践

作者：**Dave Helstroom**、**Trisha Weir**、**Evan Leonard**、**Kurt Delimon**

编辑：**Kavita Guliani**

当你让某人说出一个Google的软件工程实践成就时，他们很可能说起的是一个类似Gmail或者Google地图这样面向消费者的产品。有的人可能能够说出像Bigtable 或者Colossus 这样的基础设施项目，但是Google其实有更多的不为人知的幕后软件工程实践。这些软件工程实践

很多是来源于SRE部门的。

Google的生产环境，在某种程度上来说，是人类所建造的最复杂的机器系统之一。SRE有运维这些系统的一手经验，这使得SRE非常适合开发内部工具解决运维问题。大部分工具与维持机器的正常运行以及保证系统延迟处于较低水平有关，但是具体形式是多种多样的。有的是二进制文件的变更发布系统，也有监控系统，还有在动态服务器组合架构上建立起来的开发环境。总体来讲，SRE开发的工具是一个完整的软件工程项目，而不是一次性的脚本和小补丁。开发这些工具的SRE也需要针对内部用户的需求进行产品规划，制定未来的发展方向。

为什么软件工程项目对SRE很重要

Google生产环境的复杂程度导致了很多内部工具必须由Google自己开发，因为很少有第三方工具是按照这种复杂情况设计的。Google在软件工程方面的成功经验，显示了SRE 自行开发工具的益处。

SRE进行软件工程非常合适和有效的原因是：

- SRE组织内所拥有的Google特有的生产环境构建知识的深度和广度使得SRE工程师可以设计和实现出能够应对大规模部署，能够在灾难中优雅降级，可以和其他基础设施项目和工具良好集成的软件。

- SRE是自己工具的直接使用者，所以SRE能够深刻理解要开发工具的重点在哪里。

- 与这些工具的直接用户—其他SRE—的密切联系使得获取直接的和高质量的用户反馈变得很容易。向一个对问题和解决方案都很熟悉的内部团体发布工具可以让开发团队更快地进行迭代。内部用户一般对UI的不足和alpha版本的问题有很强的包容性。

从实践的角度来看，让具有SRE经验的工程师开发软件对Google有非常明显的好处。从组织架构设计上来说，SRE组织的成长速度要低于SRE所服务的用户服务的成长速度。SRE组织的一个指导思想是，团队大小不应该与用户服务规模成比例增长。在用户服务成指数增长的情况下，想要保持SRE团队以线性增长需要不断地进行自动化工具的开发，

以及不停地优化工具、流程，消除一切其他日常运维相关的效率问题。在这种情况下，让有直接运维经验的人来开发对应的生产工具是非常合理的。

从另外一方面来讲，每个单独的SRE，以及整个SRE组织，也会从这些SRE驱动的软件工程项目中获益。

完整的软件工程项目在SRE组织内部提供了一个职业发展的方向，也提供了一些磨炼编程技能的良好机会。长期的软件工程项目开发可以在on-call轮值之余提供平衡工作的选择，可以为同时想保持软件工程技能与系统工程技能的工程师提供一个满意的工作机会。

在减轻其他SRE工作的压力之外，这些软件工程项目更可以为SRE组织吸引和留住拥有很多不同技能的工程师。SRE团队最需要的就是技能的多样性，成员多元化的背景和多样化的解决问题的方式可以避免在团队中出现盲点。为了实现这个目标，Google一直强调要给SRE团队同时配备具备传统的软件工程经验的工程师和具备系统工程经验的工程师。

Auxon 案例分析：项目背景和要解决的问题

这篇案例分析是关于Auxon的，这是SRE内部开发的一个自动化容量规划的工具。要理解Auxon是如何诞生的，以及它能解决的问题，我们首先来看一下容量规划要解决的一些问题，以及传统方法的（Google内部的以及行业内普遍采用的）不足之处。有关这里使用的服务和集群的定义，请参看第2章。

传统的容量规划方法

计算资源容量的规划有很多方法（参见文献[Hix15a]），但是大部分方法都可以简化为下列这些：

1.收集对未来项目需求的预测

需要多少资源？这些资源什么时候需要，以及它们需要在什么物理位置？

- 使用我们今天拥有的最佳数据来计划明天。

- 预测长度一般是几个季度到几年。

2.制定资源的采购、构建和分配计划

基于上述预测，我们如何能最好地满足未来的资源需求？需要在“哪里”构建“多少”资源？

3.评审，并且批准这个计划

这项计划是不是合理的？这项计划是否和预算相符，是否符合产品的期望与技术的要求？

4.部署和配置对应的资源

一旦资源最终到位（有可能是在一段时间内），哪些服务最终会使用该资源？如何能够将底层的CPU、磁盘等资源合理配置给服务使用？

这里要着重强调的是，容量规划是一个永远没有尽头的循环：我们的假设时刻在变化，资源部署可能会延期，预算可能会改变，我们制定的计划也会一再变化。而该计划的每次变化，都必须向下传递到未来的计划中。举例来说，本季度遇到的一个资源短缺问题，必须要在未来的某个季度中得到补偿。传统的容量规划使用“需求”作为一个关键驱动值，在每次“需求”有变动时，手工调整供给，以满足“需求”。

不可靠性

传统的容量规划过程容易产生出一个非常不可靠的资源配给计划，该计划会由于出现某些看起来很小的改动而全盘失效，例如：

- 该服务可能出现了效率下降的问题，从而需要更多的资源以满足同样的业务需求。• 该服务变得更受欢迎，用户“需求”增加，导致资源的需求也随之增加。

- 某个新计算集群的上线日期推迟。

- 与性能有关的某个产品设计决策变化导致服务的部署规模改变，

从而导致资源需求改变（例如，产品决定每个视频需要存两份，而不是一份，将会导致资源用量大幅变化）。

通常，对细节的小改动需要重新整合资源的配置计划，以确保该计划仍然有效。而稍微大一点的改动（例如资源到位的时间推迟和产品策略的改变）基本就会导致整个计划推翻重来。某单个集群的资源延迟到位会影响到多个服务全局的冗余程度以及延迟要求受到影响，这种类型的变化必须要体现在容量规划中。

同时，我们还要考虑的是，每个季度的容量规划（或者其他时间间隔）都是基于前一个季度的容量规划制定的，这意味着每个季度的执行计划变化必须要更新未来季度的规划。

耗时巨大，同时不够精确

对很多团队来说，收集足够的数据以预测未来需求的过程是非常麻烦以及容易出错的。当需要寻找合适的资源以满足未来需求的时候，又面临着选择合适资源的问题。举例来说：如果产品在延迟方面的要求意味着某服务必须要用部署在相同大陆上的资源服务用户，那么在北美洲的空闲资源就无法用于满足亚洲的容量问题。每种对未来的预测都包括某种“限制”，限制了什么条件的资源能够满足这个需求，这种限制是跟产品意图紧密相关的，下一节我们会详细叙述。

将带有“限制”的资源请求与实际可用的资源进行结合也是一个很麻烦的过程。手工将资源的需求编排（bin-pack）到可用的资源过程中非常复杂和烦琐，尤其是还要考虑到预算的情况。

这个过程看起来已经很糟糕了，更糟的是，这个过程采用的工具常常是非常不可靠，或者非常难用的。电子表格（spreadsheet）经常遇到扩展性的问题和错误检查能力有限。数据可能会过期（stale），而对改变的追踪常常是很困难的。这常常导致制定计划的团队不得不大幅简化模型，以降低复杂度，只为了能够勉强满足他们的容量规划需求（而无法做到最优化配置）。

当某个服务的规划者需要将一系列容量需求与可用资源对应起来的

时候，不精确性是很要命的。最佳压缩（bin-packing）问题是一个NP-hard 问题，不适合人类进行手工计算。更重要的是，在这个阶段中，每个服务所提供的容量请求常常是非常死板的，如在集群Y中需要X个CPU。但是为什么需要X个CPU，以及为什么在Y集群中需要的原因已经早已湮灭在信息的传递过程中了。

这样的后果是，我们要消耗大量的人力才能产生一个勉强可用的、极不精确的资源配置结果。这个过程非常不可靠，没有什么好的方法可以保证产生出一个优化结果。

解决方案：基于意图的容量规划

列出你的要求，而不要拘泥于具体实现细节。

Google内部的大部分团队都已经切换到了这套我们称为基于意图的容量规划流程。简单来说，这个规划过程是将服务的依赖和资源的参数（也就是意图）用编程的方式记录下来，同时利用一个算法自动产生资源的配给方案，包括在哪个集群中将多少资源配置给哪个服务这些细节。如果需求、供给，或者某个服务的产品需求发生变化，我们可以随时重新生成这项计划，以便更好地分配资源。

由于这里记录了每个服务的真实需求和它们的选择自由度信息，所产生的容量规划在面临变动的时候非常可靠，我们可以产生出一个满足最多要求的最优解。由计算机来处理最佳情况的计算能够节省大量的人力，从而使得服务负责者可以关注在更高级的目标，比如SLO的保障，生产系统的依赖问题，以及服务基础架构的优化上，而不是费力抢夺底层的资源。

另外一个好处是，用计算优化的方式来将容量的需求和实际供给对应，可以提供更高的计划精确度，最后可以给整个组织降低更多的成本。最优解的计算仍然并不能称为完全解决，因为某些配置问题仍具有NP-hard的计算复杂度。但是我们现在的算法已经能够找到一些已知的局部最优解。

基于意图的容量规划

“意图”是服务负责人对如何运维该服务的一个理性表达。从具体的容量需求到背后理性原因的表达通常需要跳过几个抽象级别。举例如下：

1.我需要50个CPU的资源，必须在集群X、Y、Z中，为服务Foo使用。

这是一个具体的资源请求，但是，为什么我们需要这么多资源，同时一定要在这三个集群中？

2.我需要50个CPU的资源，在地理区域YYYY中的任意三个集群中，为服务Foo使用。

这项请求增加了更多的选择自由度，也更容易被满足，但是仍然没有解释这项请求背后的原因，为什么我们需要这么多资源，以及为什么需要三个集群？

3.我想要满足Foo在每个地理区域的需求增长，同时保障N＋2冗余度。

现在有了更多的选择自由度，同时我们可以更好地理解如果Foo没有获得相应的资源，究竟会造成什么后果。

4.我们想要将Foo以99.999%的可靠度运行。

这是相对更抽象的一个需求，当容量得不到满足时的后果也更清楚了：可靠性会下降。而且我们从中获得了更多的选择自由度：也许以N＋2运行并不是足够的，或者并不是最优化的情况，其他的某种部署计划是更合适的。

所以，我们应该在容量规划过程中采用哪个级别的抽象呢？理想情况下，所有上述级别都应该包括，服务提供的“意图”越多，它们得到的好处也就越大。Google的经验告诉我们，一般的服务在达到第三步的时候获得的好处最大：提供足够的选择自由度，同时可以将需求得不到满足的后果用高阶的、易理解的方式表达。某些特别复杂的项目可能需要向第四步努力。

表达产品意图的先导条件

完整表达某个服务的意图需要哪些信息？我们需要依赖关系、性能指标和优先级信息。

依赖关系

Google的服务在运行时需要依赖很多其他基础设施和服务，这些依赖服务的可用信息会极大影响某个服务的位置选择。举例来说，假设我们有一个最终用户可用的服务Foo，依赖某个基础设施存储服务Bar,Foo要求Bar服务必须处于30ms的网络延迟范围之内。这项要求对决定Foo和Bar的位置选择极为重要，基于意图的资源规划过程必须将这条限制纳入考虑范围。

更为重要的是，服务依赖是嵌套的，继续上面的例子来说。假设服务Bar要依赖服务Baz，一个底层的分布式存储系统，和Qux，某个应用程序管理系统。我们需要考虑Bar、Baz、Qux来决定在哪里放置Foo。有的时候不同系统可能依赖同一个系统，但是产品意图不同。

性能指标

某个服务的需求最后可以分解为对一个或更多其他服务的容量需求。对整个依赖链的理解可以帮助我们规划出大致的需求压缩计划，但是我们仍然需要更多的信息以决定预期的资源用量。Foo服务需要多少计算资源以服务N个用户请求？我们需要Bar服务提供多少Mb/s的数据以服务N个Foo服务的用户请求？

性能指标是依赖关系中的黏合剂。它们将一种或多种高阶资源类型转换成低阶资源类型。我们常常需要进行负载测试和资源用量监控来获得相应的性能指标信息。

优先级

每个资源的请求都会面临无法逃避的问题，在资源不够的情况下，哪些资源的请求可以被牺牲掉？

可能Foo服务的N＋2　冗余度要比Bar服务的N＋1冗余度更重要。或

者某个功能X的上线没有 Baz服务的N＋0需求重要（意思是必须保证Baz服务有足够容量）。

基于意图的规划过程强迫这些计划变得更为透明、开放和一致。资源限制仍然需要一定程度的妥协，但是在之前的过程中，优先级划分经常是随意的，对服务负责人不够透明。基于意图的规划过程使得这些优先级可以划分得更为细致。

Auxon简介

Auxon是Google开发的一个基于意图进行容量规划的工具。同时，它也是SRE设计和研发的软件的一个良好案例：它是由SRE内部的一小组软件工程师在一个项目经理带领下耗时两年时间写就的。Auxon是SRE内部软件工程的一个完美案例。

Auxon目前正在被用来规划百万美元级的计算资源分配，它已经成为Google几个重要部门的关键性容量规划组件之一。

从产品角度上来说，Auxon为收集基于意图的服务资源要求和依赖信息提供了工具。这些用户的意图通过一系列对服务的要求来表达，例如：“我的服务必须在每个大陆有N＋2的冗余度”或者“前端服务器至多只能距离后端服务器　50ms”。Auxon将这些信息通过用户配置信息或者编程API收集起来，同时将这些人工指定的产品意图转化成机器可以使用的限制条件。这些需求条件可以指定优先级，这样在资源不够的情况下可以更好地分配资源。这些资源需求—也就是产品意图—最后会形成一个巨大的线性规划表达式。Auxon通过对该表达式求解，并且利用一系列组合的最优压缩算法最终形成一个资源分配计划。

图18-1列出了Auxon的主要组件。

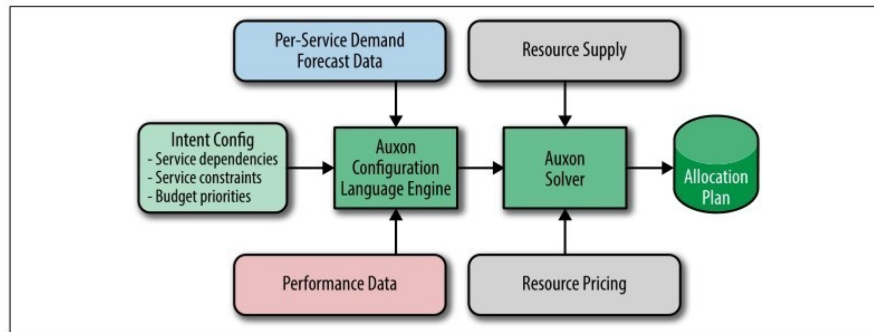


图18-1:Auxon的主要组件

数据信息（Performance Data）描述了某个服务的规模化能力。针对集群Y中的需求X，需要多少单元的依赖服务Z资源？这些数据可以根据服务的成熟程度用不同的方法获得。某些服务是通过压力测试得到的，某些是从过去的性能数据中得出的。

每个服务的需求预测数据（Per-Service Demand Forecast Data）描述了针对服务的需求预测信息。某些服务根据它们需求的预测信息推断资源使用量—分大陆区域预测的QPS信息。不是所有的服务都能这样预测需求（例如存储服务Colossus），某些服务的需求仅仅来源于依赖它们的服务。

资源供给（Resource Supply）提供了基础资源的可用性，举例来说，未来某个时间点可用的物理服务器的总量。以线性规划术语来说，资源供给是上限upper bound，限制了服务的增长和放置选择。我们的终极目标是，在服务意图配置的框架内最优化资源供应的使用。

资源价格（Resource Pricing）提供了基础资源的成本信息。举例来说，物理服务器的成本可能根据所在数据中心的位置和电源成本差别很大。在线性规划术语中，这些价格代表了总体资源的成本，也是我们要最优化的“目标”。

意图配置信息（Intent Config）是向Auxon输入基于意图的信息的关键渠道。这里定义了每个服务，以及服务之间的依赖关系。这个配置文件最终会成为其他组件的黏合剂，将其他的组件结合起来。这个部分目前被设计为人类可读和可配置的格式。

Auxon配置语言引擎（Auxon Configuration Language Engine）处理从意图配置信息中获取到的信息，将信息转化为机器格式，也就是Auxon 求解器所需要的Protocol buffer格式。同时，这个组件会进行一些简单的正确性检查。这个组件的作用就是在人类可维护的配置信息与机器可用的格式之间进行转换。

Auxon 求解器（Auxon Solver）是整个工具的大脑。它根据从Auxon配置语言引擎中获得的优化请求建立起一个庞大的混合整型线性规划程序。这个工具的扩展性非常好，可以同时运行在Google集群内的几百台甚至几千台机器上进行并行求解。除了针对混合整型线性规划的求解程序之外，Auxon求解器还包括任务的分发、工作池的维护，以及决策树相关的程序。

资源分配计划（Allocation Plan）是Auxon求解器的最后产物。它描述了在何处的何种资源应该分配给哪个服务。这个计划就是基于意图的资源规划的实现细节。该分配计划同时包括了哪些资源需求无法得到满足—例如，某个需求由于资源短缺无法得到满足，或者资源选择条件太过严格而导致无法满足。

需求和实现：成功和不足

Auxon这个想法起源于一个SRE和一个技术项目经理的共同努力。他俩分别负责对应项目中的容量规划任务。经历过基于电子表格的手工容量规划之后，他们都对过程中的效率低下部分和自动化改善机会非常了解。

在Auxon的开发过程中，负责开发的SRE团队始终与生产环境的变化密切接触。该开发团队仍负责参与多个Google服务的on-call轮值，同时经常参与这些服务的技术管理层的设计讨论。通过这些持续不断的交流，整个团队可以保持与生产环境的联系：他们既是这些工具的消费者也是开发者。当工具出现问题时，该团队会直接受到影响。团队利用他们的第一手经验决定工具的功能优先级。对问题的第一手经验和了解不仅仅使得该团队具有极强的主人翁意识，并且使得该产品在SRE内部有极强的可信度和可靠性。

近似

不要过于关注完美和解决方案的纯粹性，尤其是当待解决问题的边界不够清晰时。我们应该更快地发布和迭代。

任何一个足够复杂的软件工程项目都会遇到一定程度的不确定性，要么是组件的设计方案不确定，要么是问题的解决方式不确定。Auxon在项目初期就遇到了这样的困难，因为团队成员对线性规划问题完全没有经验。而对线性规划的限制，是整个产品的核心功能，也并没有完全理解。为了解决团队对这个问题的困惑，我们一开始选择构建一个简化版的Auxon求解器（称之为傻瓜求解器），该求解器使用一些简单的启发性逻辑来处理用户的资源请求。虽然这个傻瓜求解器并不会真正返回一个最优解，但是却能让团队对该产品的远景体验有更切实的感觉。

当在开发过程中采用模拟近似手段加快开发的时候，我们还要注意保证团队在未来能对该手段进行改进，甚至重新引入新的近似手段。在开发傻瓜求解器的时候，整个求解器接口和Auxon之间是抽象分离的，这样求解器的具体实现可以随后改变，而不会影响其他的Auxon组件。最终，随着对线性规划模型有了更多了解，我们可以一次性用这个更聪明的求解器替换傻瓜求解器。

Auxon的产品需求也有很多不确定的地方。构建这样一个需求模糊的软件项目经常让人很沮丧，但是这些不确定性并不意味着项目无法推进。借助这种模糊性可以促使软件在设计和实现上变得更为通用和模块化。举例来说，Auxon项目的其中一个目标是和Google内部的其他自动化工具结合，从而使资源分配方案可以直接在生产环境上执行（分配资源、上线、下线、调整服务大小等）。但是，在设计时，这个自动化执行系统还尚未稳定，各种各样的接口都在使用中。Auxon没有发明一个特殊的模型与各种自动化工具对接，而是采用了一种自己的设计，将“资源分配计划”变得更为抽象和实用，从而使得各种自动化系统自行将其整合进来。这种实现无关的方式，使得Auxon接入新用户的流程更容易，因为它并不强迫用户换用某一种自动化工具、某一种预测工具，或者某一种性能监控工具。

同时我们还利用模块化设计来处理建立机器性能模型时的需求模糊

问题。未来物理服务器平台的性能数据（如CPU）是很少的，但是我们的用户需要以某种方式来对不同的机器使用场景进行建模。我们将物理服务器数据隐藏在一个接口之后，允许用户自己将未来的机器性能数据替换进来。后来，随着对需求进行更加清楚的定义，我们还延伸了这个模块，提供了一个简单的通用性能数据模型类库。

如果一定要从Auxon案例中抽取一个共同点，那就是说了很多次的“发布与迭代”，这条建议适用于很多SRE软件工程项目。不要等待完美的设计，而是应该按照既定的规划继续前进。当我们遇到不确定因素时，应该力求将软件设计得更为灵活，以便当未来流程或者策略有大的变动时，不需要花费很大精力就可重新实现。但是同时，也要保证所提供的通用解决方案一直有一个真实的使用案例，以确保设计可以满足真实的需求。

提升了解程度，推进采用率

正如其他产品一样，SRE研发的软件系统也需要针对目标用户和需求来设计。SRE软件项目需要靠用途、性能和对Google生产环境可靠性的提高，或者对SRE用户工作压力的降低等确实可见的优势来赢取用户的采用。如何在组织中推广该工具，同时吸引更多的人来使用，是一个项目成功的关键。

不要低估提高人们对该产品了解程度的难度—通常发一封公告邮件或者做一个简单的演示是不够的。向大型团队推广内部软件工具需要以下几点：

- 持续的和完整的推广方案。
- 用户的拥护。
- 资深工程师和管理层的赞助，因为他们看到了项目的实用潜力。

在开发过程中，时刻从用户角度出发来设计对提高可用性很重要。使用工具的工程师可能没有时间和精力去通过源代码学习如何使用一个工具。虽然内部用户相对来说比较能够容忍界面上的不足之处，适当地提供文档还是有必要的。SRE平时的工作已经很忙了，如果你的解决方

案使用起来太困难，或者太令人迷惑，SRE通常会选择自己开发了。

设立期望值

当一个在问题领域已经有多年经验的工程师设计一个产品时，我们非常容易陷入到对完美的最终产物的幻想中去。但是，给产品设计一个最小成功条件，或者是最小可行产品（MVP）是很重要的。一个工程项目如果承诺了太多、太快，而无法做到，很容易丧失可信度。同时，如果一个项目无法产生出足够有吸引力的结果，可能没有办法找到足够的人来试用。通过递进式的、稳定的、小型发布可以提升用户对项目的信心。

在Auxon案例中，我们通过平衡长期计划和短期修复来提高迭代速度。我们向用户承诺：

- 任何试图使用该产品（包括配置等）的团队，都可以马上通过这个产品消除需要手工整合短期资源请求的痛苦过程。

- 随着Auxon的新功能不断增加，同样的配置文件可以保持不变，并且能提供更多的节约长期成本的好处。这个项目计划图使得用户可以很快地确定他们目前的用例是否还没有实现。同时，Auxon的迭代式开发流程可以很快地修改开发优先级或者制定新的里程碑。

识别合适的用户

Auxon的开发团队意识到这个特定的解决方案可能不适用于所有人。很多大型团队都已经有了内部自己开发的容量规划方案，起码目前工作良好。虽然目前他们的自定义工具并不完美，但是这些团队也没有足够的动力去使用一个新的工具，尤其是当这个新工具还处于不完美状态时。

初期版本的Auxon目标用户是那些还没有容量规划流程的团队。因为这些团队不管是自己开发工具还是采用我们的都需要花费一定的时间和精力。早期的几个成功案例证实了这个项目的实用性，从而将一些早期用户变成了该工具的推广者。能够量化工具的实用程度还有更多的好处，当我们将另外一个Google部门迁移到该工具上时，团队撰写了一篇

前后流程的案例分析对比。这项案例分析中描述的流程和人力的节约使得其他团队更愿意试用Auxon。

客户支持

虽然SRE开发的软件项目主要服务于技术项目经理（TPM）和有技术能力的工程师，但任何一个足够创新的软件项目对用户来说都有一定程度的学习曲线。不要害怕在项目初期给新用户提供一定程度的帮助，以确保他们可以更好地使用该工具。有的时候自动化同时也会面对一定程度上的情绪挑战，比如有的时候用户会恐惧自己的工作被一个shell脚本所替代。通过和早期用户一对一地共同协作，我们可以更好地处理这些情绪问题，并且更好地展示这项工具的作用在于消除重复性手工劳动，而配置文件、修改的流程和最终结果仍由这些团队负责。后来的用户可以借鉴早期用户的成功经历。

更多的是，SRE团队是分布在全球的，早期成功案例的另外一个好处就是可以成为当地的专家，帮助其他当地团队试用该产品。

在合适的层次上进行设计

我们在开发过程中使用了一个新术语“不可知性”。开发一个足够通用的软件以确保各种各样的数据都能作为输入，是Auxon的一个关键理念。软件对“不可知性”的支持确保用户不需要为了使用Auxon框架而采用某种特定的工具。这种做法使得Auxon可以在更多的不同的团队开始使用时保证足够通用。我们面对潜在用户时展现的是“不管你现在有什么数据，我们都可以采用”。通过避免对一两个大型用户的过度定制，我们降低了新用户的采用门槛，同时提高了在整个组织内部的采用率。

我们同时也主动避免落入“100%”采用率的陷阱。在很多情况下，追求设计和开发针对每个项目都适用的功能是投入产出比例不划算的。

团队内部组成

在SRE内部选择团队成员进行软件开发时，我们发现最好是能够合并通用型人才（generalist）和领域专家组成一个种子团队，通用型人才可以很快地开始工作，而资深领域专家可以提供更广阔的知识和经验。

这样一个多样化的团队可以避免设计盲点。

对团队来说，尽快与必要的专家（specialist）建立良好的工作关系是很重要的，这使得工程师可以很快地在新问题上上手。对很多其他公司来说，SRE团队解决新问题时通常需要外包，甚至请一些咨询顾问。但是大型公司的SRE团队经常可以和内部专家直接合作。在构思和设计Auxon的初期，我们通过将系统设计展示给Google内部的运维研究（operation research）和量化分析（quantitate analysis）部门，获取了更多的领域知识，让Auxon团队成员更好地了解容量规划的细节。

随着项目的进行，Auxon的功能变得复杂多变，团队吸收了一些有统计学和数学优化背景的新成员。在其他小型公司内，这可能要通过寻找外部咨询顾问来做到。这些新的团队成员能够在项目的基础功能中找到可以优化的部分，这时优化已经成为我们最高优先级的目标。

每个项目都有正确的时间点来引入领域专家。通常来说，该项目应该已经有了一定程度的成功，而且在团队成员能够充分获益的情况下才是好时机。

在SRE团队中培养软件工程风气

是什么使一个简单的一次性工具可以演化为一个长期的软件工程项目呢？是一些重要的正面信号包括相关领域专家表示的兴趣，或者技术实力很强的目标用户群体（因此可以在项目早期提供非常高质量的问题报告）。该项目还应该提供足够明显的优势，例如减少SRE的琐事，优化现有的基础架构设施，或者将一个复杂流程简单化等。

一个项目的目标应该与整个组织的发展目标一致，这样技术领导层可以为该项目的影响力背书，从而在自己的团队内或者整个组织内宣传你的项目。组织内部的推广以及评审可以帮助避免相互冲突的项目或者重复的项目发生。同时，一个能够为整个组织服务的项目也会容易招募到工程师和支撑性资源。

什么样的项目是不合适的呢？和很多你在其他软件工程项目上能直观识别的红色警告一样，例如同时改变太多组件的项目，或者软件设计

要求全有或全无的上线方式，无法采用迭代式开发的项目。因为Google SRE项目目前是以所服务的项目为核心进行组织的，SRE主导的开发项目非常容易过于专注某一个服务，而仅仅对整个Google的一小部分提供帮助。因为团队的关注点在于提高所服务产品的用户体验，SRE项目经常不能做到足够通用以被其他的SRE团队采用。另外一方面，过于通用的框架也会带来问题。如果某个工具太过专注于灵活或者通用，很可能无法针对任何一个具体案例产生足够的价值。如果一个项目目标太大、太抽象，很可能需要投入非常大的研发力量，但是却没有足够可靠的实际案例来证明自己的实用性。

一个使用范围很广泛的例子是：Google SRE开发的一个三层负载均衡器非常成功，以至于最后被改造成了直接对用户提供负载均衡服务的Google Cloud 负载均衡器（参见文献[Eis16]）。

在SRE团队中建立起软件工程氛围：招聘与开发时间

SRE通常是通用型人才，这种广度优先而不是深度优先的学习方式使得SRE对全局知识掌握得很多（没有什么比现代技术基础架构设施的内在工作原理涉及面更广泛的了）。这些工程师通常有很强的编程能力和软件开发能力，但是很多人却没有很多传统软件开发工程师的工作经验。他们没有产品讨论的经历，也没有思考用户需求的经历。一个早期SRE软件开发项目的一名工程师说的话，可以很好地总结SRE对软件工程的态度："我已经写好了设计文档，为什么还需要用户需求分析呢？"TPM和PM，相比之下有更多的面向用户的软件开发的经验，可以跟工程师结合，更好地培养出一个兼顾产品开发和实战经验的氛围。

专注的、没有干扰的项目开发时间是任何软件开发项目都必需的。专注的项目开发时间对推动一个项目的进展很关键，因为SRE不停地在多个任务间切换的时候，写代码几乎是不可能的，更不要说关注于大型的更复杂的项目了。因此，能够不受干扰地在一个软件工程项目上工作，是吸引工程师开发项目的一个因素。这些时间必须要时刻得到保障。

SRE内部的主要软件项目都是从一个小项目开始，随着采用率的提高变得更正式的。在这一点上，某个项目可能会分支到几个不同的方向

上：

- 停留为某个工程师的草根型项目，在业余时间内开发。
- 通过某种正式内部渠道成为一个正式项目（见下一节）。
- 从SRE领导层获得专属的认可，以扩展成为一个完全独立的软件开发项目。

然而，在上述任何一个场景中，进行开发的SRE仍然要继续作为SRE工作，而不是变成嵌入在SRE体系中的全职开发人员。通过对生产环境保持一定程度的熟悉，SRE可创造出更好的工具。因为他们既是工具的创造者，也是使用者。

做到这一点

如果你喜欢SRE体系内部也进行软件开发这个主意，那么你可能正在思考如何将软件开发体系引入专注于服务运维体系的团队中。

第一，我们应该认识到这个目标是一个技术问题，更是一个组织架构问题。SRE习惯于和他们的团队成员紧密合作，快速分析和应对问题。因此，软件开发其实是违背了SRE的快速写一些代码满足当下需求的工作习惯的。如果SRE团队很小，这种方式可能还没有那么多问题。但是，随着组织的增大，这种一次性的代码可能当时确实有用，但是适用范围特别窄或者用途单一的一次性软件开发方案无法在团队中共享，从而导致很多重复劳动和时间的浪费。

第二，我们要考虑在SRE中进行软件开发真正要实现的目标是什么。我们是想在团队中推广更好的软件开发实践，还是想通过软件开发得出一些可以在团队间共享的自动化方案，甚至成为组织内部的标准？在大型团队中，后面这种标准化需要一定时间，甚至延续数年。这样一种变化需要同时在多个方面推进，但是经常有很好的回报率。以下是Google的一些指导经验。

创建并且宣扬一个明确的信息

定义和宣扬你的战略目标和计划很重要。同时最重要的是，要宣传

SRE团队能够从中获得的好处。SRE是一群天生的怀疑论者（事实上，怀疑论是我们在招聘的时候重点选拔的一类特性）。SRE对该软件工程的第一反应可能是："这听起来成本过高"（too much overhead），或者是"这样是不可行的"（will never work）。对待这种情况，我们需要创造出一些可靠的案例展示这些项目如何帮助SRE：

- 该软件方案能够持久地对新SRE员工起到加速作用。

- 减少同样一个任务的执行方式，使得整个组织可以从某个单独团队的成果中获益，以便促进信息流通。

当SRE开始问你这个方案"如何"工作，而不是"能否"工作时，你应该知道你已经跨过了第一道难关。

评估组织的能力

SRE有很多技能，但是通常情况下，SRE团队缺少作为一个产品团队构建和交付完整产品的经验。为了能更好地开发有用的软件，我们实际上要创建一个产品团队。这个团队需要SRE团队中缺少的一些角色和技能。团队中谁来担当项目经理的角色？谁来负责向用户宣传？你的技术领导人或者项目经理有足够的经验和技能运转一个敏捷开发流程吗？

通过内部招聘来填充这些不足是一个好策略。通过咨询你的产品开发团队以辅导或者培训的方式建立起一套敏捷流程。同时通过咨询产品经理来帮助确定产品需求和优先级分配。有的时候可能需要一个专属人来负责这些角色。在有了一些正面结果的情况下招聘这些人可能会容易一些。

发布并且迭代

当你开始推动一个SRE软件开发项目时，你的工作被很多人密切关注。建立可信度的方式是在合理的时间内交付一些实用价值。第一轮的产品应该面向相对直接并且易于实现的目标—那些没有争议的，或者现成的解决方案的目标。我们同时发现，将这个方法对应一个六个月的产品更新周期可以让团队集中精力寻找正确的功能集来实现，并在实现这些功能的同时学习如何成为一个高效的研发团队。在初始发布结束之

后，一些团队迁移到了测试后立即上线（push-on-green）这样一个模型，以更快地发布新功能及获取反馈。

不要降低标准

在开发软件的过程中，我们可能不可避免地想要走一些捷径。一定要抵制住这种诱惑，用要求产品研发团队的标准来要求自己，例如：

- 问自己这样的问题：如果这个产品是其他团队开发的，我们是否会考虑使用？

- 如果你的解决方案被广泛采用，它可能会成为SRE正常工作的关键工具。因此可靠性是很重要的，该项目是否有足够的代码评审？是否有端到端的集成测试？可以找另外一个SRE团队，让他们按常规服务生产交接的标准来评价你的产品。

软件工程项目建立可信度很慢，但是失去它却非常快。

小结

软件工程项目在Google SRE组织内部随着人数增多而增加，成功与失败的案例为以后的项目铺平了道路。正如Auxon 解决复杂的容量规划问题那样，独特的一手生产环境运维经验使得SRE经常可以用创新的手段处理老问题，SRE驱动的软件开发项目对整个公司建立起可持续的运维团队来说也是很有帮助的。因为SRE团队经常开发用来简化低效流程或者自动化的工具，这些项目意味着SRE团队不需要和部署服务的规模同比线性增长。最终，SRE花在软件开发的精力会对整个公司、整个SRE组织，以及SRE个人都产生回报。

第19章 前端服务器的负载均衡

作者：**Piotr Lewandowski**

编辑：**Sarah Chavis**

Google每秒要处理数以百万计的请求，与你猜想的一样，我们是使用多台服务器同时承担这些负载的。即使我们真的有一台非常强大的超级计算机，可以处理所有这些请求（想想这个模式下面光是网络带宽的要求吧！），我们还是不会采取这种受单点故障影响的策略：运维大型系统时，将所有鸡蛋放在一个篮子里是引来灾难的最好办法。

本章关注于高层次的负载均衡—Google是如何在数据中心之间调节用户流量的。下一章会更深入地探讨我们是如何在一个数据中心内进行流量分发和负载均衡的。

有时候硬件并不能解决问题

假设，这里仅仅是假设—我们有一台非常强大的服务器，和与之配套的永不出故障、带宽充足的网络连接。这是否就能满足Google的需求了呢？并不是。就算拥有这样的一套配置，它仍然会受到一些物理条件的限制。例如，光速是通过光纤通信的制约性因素，这限制了远距离传输数据的速度。就算在一个理想的情况下，采用这样一种受单点故障影响的基础设施也是一个糟糕的主意。

在现实中，Google拥有数以千计的服务器，也同时有比这个数量更多的用户在发送请求。很多用户甚至同时发送好几个请求。用户流量负载均衡（traffic load balancing）系统是用来决定数据中心中的这些机器中哪一个用来处理某个请求的。理想情况下，用户流量应该最优地分布于多条网络链路上、多个数据中心中，以及多台服务器上。但是这里的“最优”是如何定义的呢？其实这里并没有一个独立的答案，因为这里的最优严重依赖于下列几个因素：

- 逻辑层级（是在全局还是在局部）。
- 技术层面（硬件层面与软件层面）。
- 用户流量的天然属性。

我们先来讨论以下两个常见的用户流量场景：一个简单的搜索请求和一个视频上传请求。用户想要很快地获取搜索结果，所以对搜索请求来说最重要的变量是延迟（latency）。而对于视频上传请求来说，用户

已经预期该请求将要花费一定的时间，但是同时希望该请求能够一次成功，所以这里最重要的变量是吞吐量（throughput）。两种请求用户的需求不同，是我们在全局层面决定“最优”分配方案的重要条件。

- 搜索请求将会被发往最近的、可用的数据中心—评价条件是数据包往返时间（RTT），因为我们想要最小化该请求的延迟。

- 视频上传流将会采取另外一条路径—也许是一条目前带宽没有占满的链路—来最大化吞吐量，同时也许会牺牲一定程度的延迟。

但是在局部层面，在一个数据中心内部，我们经常假设同一个物理建筑物内的所有物理服务器都在同一个网络中，对用户来说都是等距的。因此在这个层面上的“最优”分配往往关注于优化资源的利用率，避免某个服务器负载过高。

当然这个例子中使用了一个非常简化的场景。在现实中，很多其他因素也都在“最优”方案的考虑范围之内：有些请求可能会被指派到某个稍远一点的数据中心，以保障该数据中心的缓存处于有效状态。或者某些非交互式请求会被发往另外一个地理区域，以避免网络拥塞。负载均衡，尤其是大型系统的负载均衡，是非常复杂和非常动态化的。Google在多个层面上使用负载均衡策略来解决这些问题：下面这一节会讨论到其中两个。为了更切实地展开讨论，我们这里主要讨论基于TCP的HTTP请求。对无状态（stateless）服务（如基于UDP的DNS）的负载均衡和这个有所不同，但是这里讨论的大部分方式都仍然适用。

使用DNS进行负载均衡

在某个客户端发送HTTP请求之前，经常需要先通过DNS查询IP地址。这就为我们第一层的负载均衡机制提供了一个良好基础：DNS负载均衡。最简单的方案是在DNS回复中提供多个A记录或者AAAA记录，由客户端任意选择一个IP地址使用。这种方案虽然看起来简单并且容易实现，但是存在很多问题。

第一个问题是这种机制对客户端行为的约束力很弱：记录是随机选择的，也就是每条记录都会引来有基本相同数量的请求流量。如何避免

这个问题呢？理论上我们可以使用SRV记录来指明每个IP地址的优先级和比重，但是HTTP协议目前还没有采用SRV记录。

另外一个潜在问题是客户端无法识别“最近”的地址。我们可以通过提供一个anycast DNS服务器地址，通过DNS请求一般会到达最近的地址这个方式来一定程度上缓解这个问题。服务器可以使用最近的数据中心地址来生成DNS回复。更进一步的优化方式是，将所有的网络地址和它们对应的大概物理位置建立一个对照表，按照这个对照表来发送回复。但是这种解决方案使得我们需要维护一个更加复杂的DNS服务，并且需要维护一个数据更新流水线（pipeline）来保证位置信息的正确性。

当然，没有一个很简单的方案，因为这是由DNS的基本特性决定的：最终用户很少直接跟权威域名服务器（authoritive nameserver）直接联系。在用户到权威服务器中间经常有一个递归解析器（recursive nameserver）代理请求。该递归解析器代理用户请求，同时经常提供一定程度的缓存机制。这样的DNS中间人机制在用户流量管理上有三个非常重要的影响：

- 递归方式解析IP地址。
- 不确定的回复路径。
- 额外的缓存问题。

以递归方式解析IP地址会造成一定的问题，因为权威服务器接收到的不是用户地址，而是递归解析器的IP地址。这是一个很严重的问题，因为这样DNS服务只能根据递归解析器的IP地址返回一个最优方案。一个可能的解决方案是使用EDNS0 扩展协议（参见文献[Con15]），该协议在递归解析器发送的请求中包括了最终用户的子网段。这样权威服务器可以返回一个对最终用户来讲最优的回复。虽然这个协议还不是正式规范，但是这些明显的优势使得最大的DNS解析器（例如OpenDNS和Google[68]）已经开始采用了。

不仅要处理返回最优IP的这个难题，处理请求的域名服务器可能同

时需要处理几千、几万个用户的请求，范围从一个小办公室到整个大陆。举例来说，某个大型的国家级电信服务商可能在一个数据中心中运行数个域名服务器以服务他们的整个网络，但是同时该服务商可能在数个大都市区域也有网络互联。该 ISP 的域名服务器返回的一个回复可能对他们当前的数据中心来讲是最优的，却没有考虑到可能对其他用户还有更优的网络路径。

最后，递归解析器一般根据接收到的回复中的时效值（TTL）来缓存和发送这些回复。这样会造成预估某个DNS回复的用户流量影响很困难：因为一个回复可能会发送给一个用户，或者几万个用户。我们利用两种方式来解决这个问题：

- 分析流量的变化，并且持续不断地更新已知的DNS解析器的用户数量，这样可以评估某个解析器的预期影响。

- 根据数据评估每个已知解析器背后的用户的地址位置分布，以便更好地将用户转向最佳地址。

准确评估地理位置分布是非常困难的，尤其是用户分布在很广的区域时。在这种情况下，我们只能针对大部分用户的情况选择最优位置进行优化。

但是“最优位置”在DNS负载均衡的语境中，到底是什么意思呢？最直接的答案是“离用户最近的位置”。但是（先不考虑确定用户位置有多难）还应该有其他的选择条件。DNS负载均衡器需要确保它选择的数据中心有足够的容量来处理该DNS回复所带来的请求。同时，负载均衡器还要保障其选择的数据中心和网络目前都处于良好状态，否则将用户导向正在经受网络故障和供电故障的地点并不是一个很合理的做法。幸好，Google可以将权威DNS服务器和我们的全局负载均衡系统（GSLB）整合起来，该负载均衡系统负责跟踪我们的流量水平、可用容量和各种基础设施的状态。

DNS中间人带来的第三个问题是跟缓存有关的。因为权威服务器不能主动清除某个解析器的缓存，DNS记录需要保持一个相对较低的时效值（TTL）。这其实是为DNS回复的变化到达最终用户的速度设置了一

个下限。[69]不幸的是，我们除了将这个因素包含在负载均衡计算之外，并没有什么其他的应对办法。

尽管有这些不足，DNS仍然是最简单、最有效的负载均衡制度，它在用户发起连接之前就生效了。另一方面，我们清晰地看到仅仅靠在DNS里做负载均衡是不够的。我们要记住RFC 1035将DNS回复限制为512字节（参见文献[Moc87]）。[70]

DNS的尺寸限制实际上为单个DNS回复能返回的地址数量设置了上限，这个上限明显是远小于我们服务器的数量的。

要真正解决前端负载均衡的问题，我们需要在DNS负载均衡之后增加一层虚拟IP地址。

负载均衡：虚拟IP

虚拟IP地址（VIP）不是绑定在某一个特定的网络接口上的，它是由很多设备共享的。但是，从用户视角来看，VIP仍然是一个独立的、普通的IP地址。理论上来讲，这种实现可以让我们将底层实现细节隐藏起来（比如某一个VIP背后的机器数量），无缝进行维护工作。比如我们可以依次升级某些机器，或者在资源池中增加更多的机器而不影响用户。

在实践中，最重要的VIP实现部分是一个称为网络负载均衡器（network load balancer）的组件。该负载均衡器接收网络数据包，同时将它们转发给VIP背后的某一个服务器，这些后端服务器可以接下来处理该请求。

负载均衡器在决定哪个后端服务器应该接收请求时，有如下几种方案：第一种（也可能是最直接的）方案是，永远优先目前负载最小的后端服务器。理论上来说，这个方案应该可以最优化用户体验，因为请求始终会被发往最不忙的机器。但是，这个逻辑对有状态的协议就不适用了，因为在处理一个请求的过程中必须使用同一个后端服务器。这就需要负载均衡器跟踪所有经过它转发的连接，以确保同一个连接的数据包都会发往同一个后端服务器。一种替代方案是使用数据包中的某些部分

创建出一个连接标识符（connection ID）（可能使用某些数据包内的信息和一个哈希算法），使用该连接标识符来选择后端服务器。举例来说，连接标识符可以用如下算式表达：

```
id(packet) mod N
```

这里的id（）是一个函数，以数据包内容为输入，得出一个连接标识符，N是所有配置的后端服务器数量。

这样负载均衡器就不用再保存状态了，每个连接的数据包也都会发往同一个后端服务器。这就成功了吗？还没有。当某个后端服务器出现问题，需要从列表中去掉时怎么办呢？这里N突然变成了N-1，而id（packet）mod N变成了id（packet）mod N-1，基本上所有的请求都被指向了另外一个后端服务器。如果后端服务器之间不同步状态，这几乎意味着所有现存连接都要中断。这样的场景即使出现得不太频繁，对用户来说也是很不友好的。

幸运的是，的确有一种替代方案。既不需要在内存中保存所有连接的状态，也不会在单个机器出故障时重置所有连接，这就是一致性哈希（consistent hashing）算法。1997年提出的一致性哈希算法（参见文献[Kar97]）描述了一种映射算法，在新的后端服务器被添加或者删除时保持相对稳定。这种算法在后端资源变化时，最小程度地减少了对现存连接的影响。最终结果是，我们平时可以使用简单的连接跟踪机制，但是在系统压力上升时切换为一致性哈希算法，例如在处理分布式拒绝服务攻击时（DDoS）。

那么回到更大的问题上来：负载均衡器究竟是如何将数据包发往某个特定的VIP后端的呢？一个解决方案是进行一次网络地址转换（NAT）。但是这要求我们在内存中跟踪每一个连接，也就是不能提供一个完全无状态的后备机制。

另外一个解决方案是修改数据链路层（OSI模型的2层）的信息。通过修改转发数据包的目标MAC地址，负载均衡器可以保持全部上层信息不变，后端将会接收到原始的来源和目标地址信息。后端服务器可以直接发送回应给用户—这被称之为直接服务器响应（DSR）。如果用户请

求很小，而回复很大（恰如大多数HTTP请求那样）,DSR可以节约大量资源，因为仅仅只有一小部分用户流量需要穿过负载均衡器。更好的是，DSR不需要保持状态。但是使用2层信息进行内部负载均衡会导致在大规模部署下出现问题：所有的机器（也就是所有的负载均衡器和所有的后端服务器）必须可以在数据链路层相通。如果服务器数量不多，网络能够支撑这样的连接那就不是问题，但是所有的机器都需要在同一个广播域中。正如你想的那样，Google 在一段时间后，由于规模原因，已经放弃了这种方案。

我们现在的 VIP负载均衡解决方案（参见文献[Eis16]）使用的是包封装（encapsulation）模式。网络服务在均衡器将待转发的网络包采用通用路由封装协议（GRE，参见文献[Han94]）封装到另外一个IP包中，使用后端服务器的地址作为目标地址。后端服务器接收到网络包，将IP和GRE层拆除，直接处理内部的IP包，就像直接从网络接口接收到的那样。网络负载均衡器和后端不再需要共存在同一个广播域中，只要中间有路由连接即可，它们甚至可以存在不同的大陆上。

包封装是一个强有力的手段，可以为我们的网络设计和改进提供足够的灵活性。不幸的是，封装机制通常也会带来成本问题：包尺寸的增加。包封装需要一定程度的成本（IPv4＋GRE，封装需要24字节），这经常导致数据包超出可用的传输单元（MTU）大小，而需要碎片重组（Fragmentation）。

数据到达数据中心内部之后，我们可以在内部采用更大的MTU来避免碎片重组的发生，但是这种做法需要网络设备支持。就像很多东西一样，负载均衡表面上听起来很简单—尽早进行负载均衡，以及分级多次进行—但是不管是在前端负载均衡方面，还是在数据中心内部都存在许多实现细节问题。

第20章 数据中心内部的负载均衡系统

作者：**Alejandro Forero Cuervo**

编辑：**Sarah Chavis**

本章关注于数据中心内部的负载均衡系统，我们将讨论在数据中心内部采用的负载均衡算法。本章覆盖了将请求路由给某个具体服务器的应用级别策略。底层的网络链接和设备（例如交换机、数据包路由等），以及数据中心级别的负载均衡都不在本章讨论范围之内。

假设目前已经有一些请求抵达了数据中心—这些请求可能来自数据中心内部，或者来自另外一个数据中心，或者两者皆有。该请求的速率还没有超过数据中心的处理能力，或者只是刚刚超过（这时数据中心级别的负载均衡还没有迁走这部分流量）。同时，我们假设数据中心中有对应该请求的用户服务，这些服务是由一些同质的、可互换的软件服务器，以进程的方式运行在很多不同的物理服务器上构成的。一个最小的这样的服务通常有至少三个这样的进程在运行（如果进程少于三个，通常意味着一台物理机器失败将会导致50%以上的服务容量不可用），最大的服务可能由超过10,000个进程组成（取决于数据中心大小）。通常情况下，服务一般由100 ~ 1,000个进程组成。我们将这些进程称之为后端任务（或者后端，backend）。其他任务，我们称之为客户端任务（client），和这些后端任务建立连接。客户端任务在处理每个请求时，必须决定哪个后端任务应该处理该请求。客户端与后端的通信采用的是建立在TCP和UDP之上的应用层协议。

我们这里要提到的是，Google数据中心内部运行了大量的、非常不同的用户服务。这些服务每个都在使用本章讨论的算法的不同组合。我们这里举的例子，和任意一个服务都不完全一样。这里采用的是一个通用化的场景，这样可以更容易地讨论对不同服务适用的各种不同算法。某些算法可能更适合（或者更不适合）某一个特定的案例，但是这些算法都是数个Google工程师在多年内积累而成的。

这些算法在Google整个技术栈的多个部分上都有应用。例如，大部分的外部HTTP请求都会由 Google 前端服务器（GFE）处理 — Google的反向代理系统。GFE使用这些算法，以及第19章中介绍的算法，将请求转发给某个特定的服务进程以处理该请求。这套系统使用一个基于URL模式匹配的配置文件将某个特定请求转发给其他团队控制的后端服

务器。为了处理这些请求（这里的回应会先回复给GFE，再转发给客户端浏览器），这些后端服务器都需要采用同样的负载均衡算法联系其他基础设施服务，或者它们所依赖的其他服务。有的时候，整个依赖处理栈会非常深，单个HTTP请求可能会触发一长串的后端依赖请求，也可能在多个节点处拓展为多个并发请求。

理想情况

在理想情况下，某个服务的负载会完全均匀地分发给所有的后端任务。在任何一个时间点上，最忙和最不忙的任务永远消耗同样数量的CPU。

在最忙的任务达到容量限制之前，我们可以继续将用户流量发往这个数据中心。正如图20-1列举的同一时间周期内的两个不同场景。在此时，跨数据中心负载均衡算法必须避免再给该数据中心发送额外请求，因为这样做可能会导致某些任务过载。

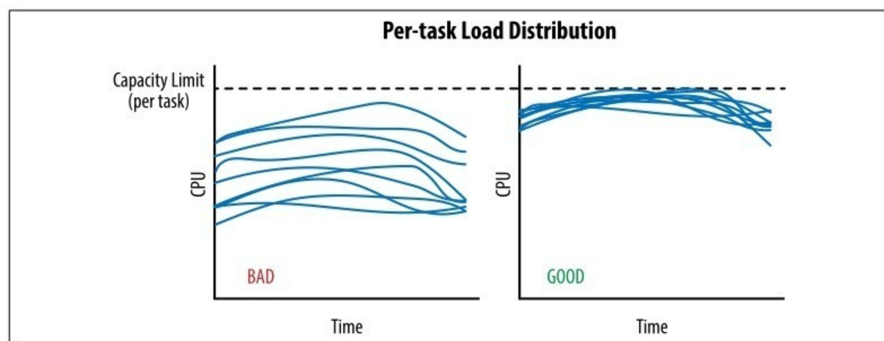


图20-1：任务负载分布的两种场景

图20-2的左图所示的情况是大量的容量都被浪费了：除了最忙的任务之外，其他任务的CPU都处于闲置状态。

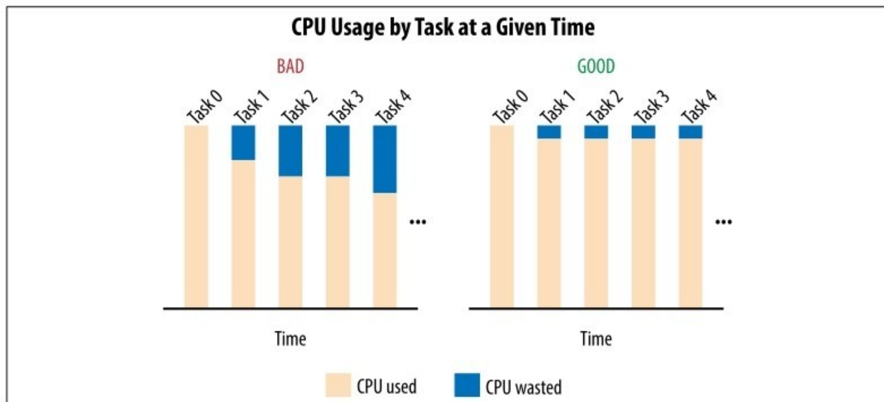


图20-2:CPU的使用和浪费的直方图

我们更正式地定义CPU[i]为某个任务i在某个时刻所消耗的CPU资源数量，假设任务0是负载最高的任务。那么，在分布情况下，我们这里消耗的CPU资源就是其他任务与任务0的CPU差值。也就是说，我们总共会浪费CPU[0]–CPU[i]的总和。在这里，浪费的意思是指　“保留的”但是“未使用”的。

上述这个例子描述了糟糕的负载均衡策略会导致资源可用率下降：你可能在数据中心中为你的服务一共预留了1000个CPU，但是却无法让实际使用超过700个CPU。

识别异常任务：流速控制和跛脚鸭任务

在我们决定哪个后端任务应该接受客户端请求之前，首先要先识别—并且避开—在后端任务池中处于不健康状态的任务。

异常任务的简单应对办法：流速控制

假设我们的客户端任务会跟踪记录发往每个后端的请求状态。当某个后端的活跃请求达到一定数量时，该客户端将该后端服务器标记为异常状态，不再给它发送请求。对大多数后端任务来说，100是个合理的限制。在大多数情况下，请求通常能够在很短的时间内完成，使得这个限制在正常情况下几乎不会触发。这种（非常简单的）流速控制机制也是一种非常简单的负载均衡机制：如果某个后端任务过载了，请求处理开始变慢，客户端会自动避开这个后端，从而将任务分配给其他的后

端。

不幸的是，这种简单的方法只能保护后端任务不受非常极端的过载情况影响，后端很有可能在该限制到达之前就过载了。反之亦然，在某些情况下，客户端可能在后端还有充足资源的情况下就触发了这个限制。例如，有些后端服务器可能有特殊的长连接请求，不会很快回复。我们曾经见到过这样的案例，这个默认限制导致了全部后端任务都被标记为不可用，所有的客户端请求都被阻挡了，直到这些请求超时并且失败。可以通过调节这个限制来暂时避免这种情况发生，但是这并不能解决底层的根源问题：识别一个任务是真的处于不健康状态，还是仅仅回复有点慢。

一个可靠的识别异常任务的方法：跛脚鸭状态

从一个客户端的视角来看，某个后端任务可能处于下列任一种状态中：

健康

后端任务初始化成功，正在处理请求。

拒绝连接

后端任务处于无响应状态。这可能是因为任务正在启动或者停止，或者是因为后端正处于一种异常状态（虽然很少有后端任务在非停止状态下停止监听端口）。

跛脚鸭状态

后端任务正在监听端口，并且可以服务请求，但是已经明确要求客户端停止发送请求。

当某个请求进入跛脚鸭状态时，它会将这个状态广播给所有已经连接的客户端。但是那些没有建立连接的客户端呢？在Google的RPC框架实现中，不活跃的客户端（没有建立TCP连接的客户端）也会定期发送UDP健康检查包。这就使跛脚鸭状态可以相对较快地传递给所有的客户端—通常在一到两个RTT周期内—无论它们处于什么状态下。

允许任务处于这种半正常的跛脚鸭状态的好处就是让无缝停止任务变得更容易，处于停止过程中的任务不会给正在处理的请求返回一个错误值。能够无影响地停止一个活跃的后端任务可以让处理代码推送、设备维护活动，和机器故障问题导致的任务重启变得对用户透明。这个停止过程通常按照以下步骤进行：

1.任务编排系统发送一个SIGTERM信号给该任务。

2.后端任务进入跛脚鸭状态，同时请求它的所有客户端发送请求给其他后端任务。这通过SIGTERM信号处理程序中调用RPC实现中的API完成。

3.任何在后端进入跛脚鸭状态时正在进行的请求（或者在进入状态之后，但是其他客户端收到通知之前）仍会继续进行。

4.随着请求回复被发送回客户端，该后端任务的活跃请求逐渐降低为0。

5.在配置的时间过后，该后端程序要么自己干净地退出，要么任务编排系统主动杀掉它。该时间应该被设置为一个足够大的值，以便一般的请求可以有足够的时间完成。每个服务的该数值都不同，一般来说取决于客户端的复杂程度，10s到150s是一个不错的选择。

这个策略使客户端可以在后端程序进行耗时较长的初始化过程中（这时后端程序还不能服务请求）就建立连接。如果后端程序等到服务可以接受请求的时候才建立链接，就增加了一些不必要的延迟。一旦后端程序可以提供服务了，它就会主动通知所有客户端。

利用划分子集限制连接池大小

在健康管理之外，负载均衡另外要考虑的一个因素就是子集划分：限制某个客户端任务需要连接的后端任务数量。

我们的RPC系统中的每个客户端都会针对后端程序维持一个长连接发送请求。这些连接通常在客户端启动的时候就建立完成，并且保持活跃状态，不停地有请求通过它们，直到客户端终止。另外一个方案是针

对每个请求建立和销毁后端连接，这样会带来极大的资源成本和造成延迟问题。在极端情况下，如果某个连接闲置时间非常长，我们的RPC实现可以自动将该连接转为“不活跃”状态，转为UDP模式连接，而非TCP模式。

每个连接都需要双方消耗一定数量的内存和CPU（由于定期健康检查导致）来维护。虽然这个消耗理论上很小，但是一旦数量多起来就可能变得很可观。子集化可以避免一个客户端连接过多后端任务，或者一个后端任务接收过多客户端连接。

选择合适的子集

子集的选择过程由确定每个客户端任务需要连接的后端数量—子集大小—和选择算法决定。我们通常使用20～100个后端数量作为子集大小，但是每个系统的“正确”子集大小主要由你的服务决定。例如，以下情况可能需要一个相对大的子集数量：

- 客户端数量相比后端数量少很多。在这种情况下，你希望每个客户端都有足够的后端任务可供连接，以免某些后端接收不到请求。

- 某个客户端任务经常出现负载不平衡的情况（也就是说，某个客户端会比其他的客户端发送更多的请求）。这个场景在某个客户端偶尔发送突发性请求的时候很常见。这些客户端任务接收其他客户端任务发来的请求，将其拓展为一个非常大的并发请求（例如，读取某个用户的全部关注者的全部信息）。因为这样的突发性请求会集中发送给该客户端任务的后端子集，我们需要更大的子集尺寸以便能够更好地将这些请求分发给更多的后端任务。

当确定子集大小之后，我们需要使用一个算法来定义每个客户端任务使用的后端子集。这可能看起来很容易，但是在一个需要高效利用资源，并且不可避免重启的大型系统中就很困难了。

选择算法应该将后端平均分配给客户端，以优化资源利用率。举例来说，如果该算法将导致某个后端过载10%，那么整个集群都需要过度分配10%的容量。这个算法也应该可以自动处理重启和任务失败，持续

不断地均衡后端任务，同时避免大幅变动。在这里，“大幅变动”指的是后端替换的过程。例如，当某一个后端进程不可用时，客户端需要临时选择一个替换后端。当这个替换后端被选中时，客户端必须建立新的TCP连接（同时可能还要进行应用级别的交互），这些都带来了额外的成本消耗。同样的，当一个客户端程序重启时，它可能要重新与所有的后端建立连接。

该算法应该同时处理客户端程序和后端程序集群的大小调整，避免对现有连接的大幅变动，同时在无法预知这些具体数字的情况下做到这一点。该功能在整个客户端集群或者整个后端任务集群滚动重启（例如升级时）的时候尤为重要（也很难正确实现）。在后端任务滚动重启时，我们需要客户端任务持续服务，对重启透明，同时连接的变动最小。

子集选择算法一：随机选择

一个最简单的子集选择算法是让所有客户端任务将后端列表随机排列一次，同时将其中的可解析/可服务状态的后端提取出来。一次性随机排列并顺序选取可以很好地处理重启和任务失败情况（在这些情况下连接变动很小），因为这种算法限制了所考虑的后端的数量。但是我们发现，这个算法在大多数实际应用场景中效果非常差，因为负载非常不均衡。

在Google负载均衡系统设计之初，我们实现了这种随机子集算法，并且计算了在各种情况下的负载均衡情况。举例如下：

- 300个客户端
- 300个后端
- 30%的子集大小（每个客户端连接90个后端）

如图20-3所示，负载最低的后端程序只有平均负载值的63%（57个连接，平均值为90个连接），而负载最高的后端程序是平均负载值的121%（109个连接）。在大多数情况下，30%的子集大小已经比我们想要使用的值要大得多。每次计算的负载分布情况都会变化（因为有随机

因素），但是总体趋势保持不变。

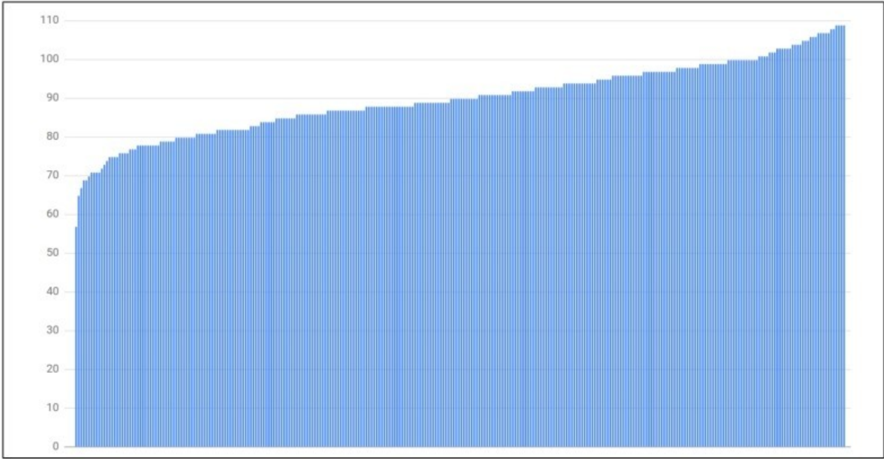


图20-3:300个客户端、300个后端和子集大小为30%的连接分布图

更不幸的是，子集大小降低时，负载会更为不平衡。图20-4 描述了如果子集大小降低为10%（每个客户端连接30个后端）的情况。在这种情况下，最低负载的后端只接受平均值的50%（15个连接），而最高负载的后端接受了平均值的150%（45个连接）。

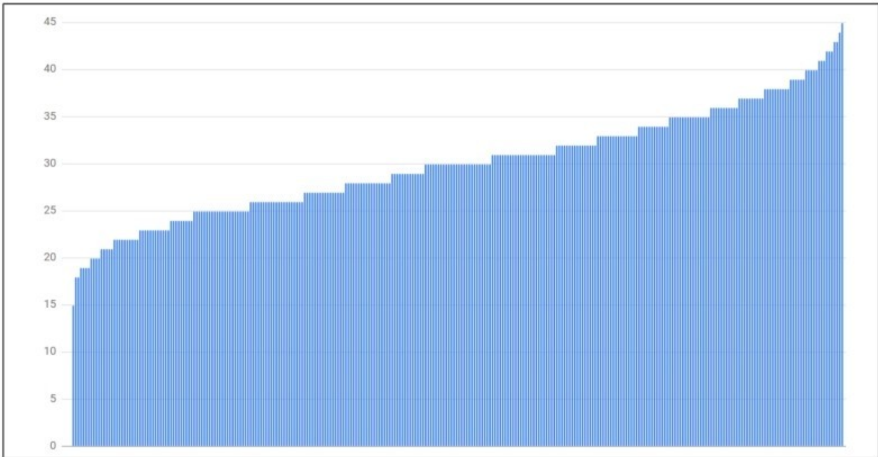


图20-4:300个客户端、300个服务器端和子集大小为10% 的连接分布图

我们最终的结论是，如果使用随机子集分布算法，要相对地平均分配负载，需要至少75%的子集大小。这样大的子集是不可行的：分布如此不均衡的算法在大规模部署情况下是不适用的。

子集选择算法二：确定性算法

Google对随机算法限制的解决方案是：确定性算法，以下这段代码实现了这个算法，详细解释在后文：

```
def Subset(backends, client_id, subset_size):
  subset_count = len(backends) / subset_size

  # 将客户端划分为多轮，每一轮计算使用同样的随机排列的列表。
  round = client_id / subset_count
  random.seed(round)
  random.shuffle(backends)

  # subset_id 代表了目前的客户端
  subset_id = client_id % subset_count

  start = subset_id * subset_size
  return backends[start:start + subset_size]
```

我们将客户端任务划分在多“轮”中，每一轮i包含了subset_count个连续的客户端，从客户端subset_count * i开始，同时subset_count是子集的个数（也就是后端数量除以想要的子集大小）。在每一轮计算中，每个后端都会被分配给一个而且仅有一个客户端任务（最后一轮除外，这时可能客户端数量不够，所以有些后端没有被分配）。

例如，我们有12个后端任务[0,11]，想要的子集大小为3，我们会进行三轮计算，每轮4个客户端任务（subset_count＝12/3）。如果有10个客户端任务，那么前述算法可能会产生如下结果：

- Round 0:[0,6,3,5,1,7,11,9,2,4,8,10]
- Round 1:[8,11,4,0,5,6,10,3,2,7,9,1]
- Round 2:[8,3,7,2,1,4,9,10,6,5,0,11]

这里要注意的关键点是，每一轮计算仅仅将整个列表中的每一个后端分配给唯一一个客户端（除了最后一轮，可能没有足够的客户端分配）。在这个例子中，每个后端任务都会被分配给2个或3个客户端任务。

这个列表应该被随机重排，否则客户端将会被分配给一组连续的后

端任务，这些后端任务可能同时进入临时不可用状态（例如，因为后端任务正在从第一个任务到最后一个任务滚动更新）。不同的轮应该使用不同的随机重排种子。如果没有变化，那么当一个后端出现问题时，它的负载将会被分配给它所在子集的剩余后端中。如果剩余后端中再有其他的故障，负载会继续叠加，从而使情况变得更严重：如果某个后端子集中有N个后端出现故障，那么它们对应的负载将会分布给其他的（subset_size – N）个后端。在每一轮计算中使用不同的随机排列是均衡负载的更好方式。

当我们在每轮计算中使用不同的重排列表时，在同一轮中的客户端会使用同一个列表，但是不同轮中的客户端会使用不同的列表。接下来，算法将子集用重排过的列表和子集大小来定义，如：

- 子集[0]＝重排列表[0]到 重排列表[2]
- 子集[1]＝重排列表[3]到 重排列表[5]
- 子集[2]＝重排列表[6]到 重排列表[8]
- 子集[3]＝重排列表[9]到 重排列表[11]

这里每个重排列表是每个客户端对应的轮中的新创建的列表。将某个子集被指派给某个客户端任务时，我们可以取该客户端在该轮中的位置对应的子集（对客户端i来说，在4个子集的情况下就是（i % 4））。

- 客户端[0]，客户端[4]，客户端[8]会使用子集[0]
- 客户端[1]，客户端[5]，客户端[9]会使用子集[1]
- 客户端[2]，客户端[6]，客户端[10]会使用子集[2]
- 客户端[3]，客户端[7]，客户端[11]会使用子集[3]

因为在不同轮的客户端会使用不同的重排列表（子集也就不同），而同一轮的客户端也会使用不同的子集，所以整个连接负载会均匀地分布。当后端总数不能被子集大小所整除时，我们允许少部分子集比其他的稍大。但是大部分情况下，每个后端所分配到的客户端数量相差最多

不超过1个。

如图20-5所示，前述案例中的300个客户端每个连接300个后端中的10个，在这个算法下结果非常好：每个后端都接收到同样数量的连接。

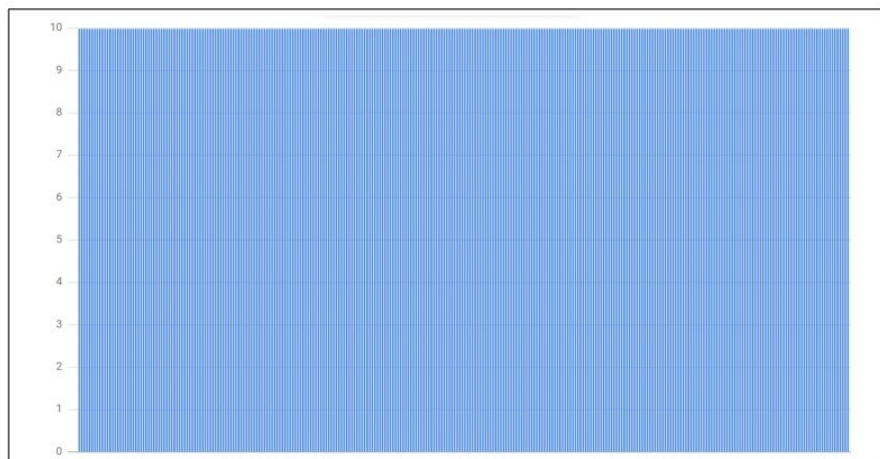


图20-5:300个客户端，300个后端，子集数量10，采用确定性算法的结果

负载均衡策略

现在我们已经知道了某个客户端任务是如何保持一定数量的健康后端的连接的，下面来看一下负载均衡的策略，也就是客户端任务用来选择它的子集中哪个任务需要接收某个用户请求的算法。大部分负载均衡策略的复杂度都来自于决策的分布式特性，每个客户端需要实时决定（利用部分甚至可能是过期的后端信息）哪个后端需要服务哪个请求。

负载均衡策略可以是非常简单的，不考虑任何后端状态的算法（例如，轮询），或者是基于后端状态的算法（例如最小负载轮询，或者带权重的轮询）。

简单轮询算法

一个非常简单的负载均衡策略是让每个客户端以轮询的方式发送给子集中的每个后端任务，只要这个后端可以成功连接并且不在跛脚鸭状态中即可。很长时间以来，这都是我们最普遍使用的策略，现在很多服务仍在使用。

不幸的是，虽然轮询策略（round robin）非常简单，并且比随机选择后端效果要好，但该策略的负载均衡结果可能很差。这里的实际数据取决于很多因素，例如不同的请求处理成本和物理服务器的差异等，但是我们发现轮询策略可能会导致最高负载任务比最低负载任务使用2倍的CPU资源。这样的负载均衡结果是非常浪费的，这是由很多因素造成的，包括以下几项：

- 子集过小
- 请求处理成本不同
- 物理服务器的差异
- 无法预知的性能因素

子集过小

一个最简单的轮询负载均衡策略表现糟糕的原因就是所有的客户端可能不会以同样的速率发送请求。在多个进程共享某个后端时，不同客户端请求速率不同是非常常见的，处于发送请求较多的客户端的子集中的后端，负载就会越高。

请求处理成本不同

很多服务的请求处理所需要的资源是不同的。在实际工作中，我们发现在很多服务中，成本最高的请求相比成本最低的请求会多消耗1000倍（甚至更多）的CPU资源。在请求处理成本无法事先预知的情况下，轮询策略就更难均衡负载了。例如，一个“返回用户XYZ昨天收到的所有邮件”的请求可能成本很低（如果该用户每天接收邮件数量很少），也可能成本非常高。

在一个请求处理成本差异性很大的系统中进行负载均衡是非常困难的。这时候有可能需要调节服务接口以控制每个请求所执行的工作数量。例如，在上述邮件查询服务中，可以引入一个分页接口，同时将该请求更改为“返回最近100封（或更少）用户XYZ昨天收到的邮件。”但是，这种语义改变通常是很困难的。这不仅需要所有客户端代码的改

变，还需要考虑额外的一致性因素。例如，该用户可能在客户端分页查询的时候收到了新的邮件，或者正在删除邮件。在这种情况下，客户端如果只是简单地枚举每个结果，并将结果结合在一起（而不是在一个固定视图中查询）可能会造成视图不一致的情况，有些消息会重复出现，而有些会被漏掉。

为了保证服务接口（以及它们对应的实现）简单，服务经常允许成本最高的请求消耗100、1,000甚至10,000倍的资源。然而，这种多变的资源要求会导致某些不那么幸运的后端偶尔接收到比其他后端成本更高的请求。在这种情况下，负载均衡的效果就取决于成本最高的请求。例如，我们的某个Java后端程序大概每个请求消耗15ms的CPU时间，但是某些请求可以轻易消耗10s。每个后端任务都预留了多个CPU内核资源，这样可以让某些请求并行处理，以降低延迟。但是当该后端接收到一个这样的大请求时，它的负载会显著上升一段时间。这时，一个异常状态下的任务可能会超出内存限制而被杀掉，或者完全停止响应（由于内存不够）。但是就算在正常情况下（后端有足够资源处理请求，同时负载会在大请求结束后恢复正常），其他请求也会由于资源竞争原因导致延迟上升。

物理服务器的差异

另外一个简单轮询策略的问题是，同一个数据中心中的所有物理服务器并不都是一样的。某个数据中心可能有不同CPU性能的物理服务器，所以，同样的请求可能对不同的物理服务器来说负载不同。

处理物理服务器的差异—也就是不需要强同质性—是Google很长时间的一个难题。理论上来说，对待繁杂的资源容量的解决方案是很简单的：只要将CPU预留值根据CPU/物理服务器类型来倍增就行了。但是，在实际实践中，要实现这个方案，需要我们的任务编排系统跟踪某个任务平均机器性能的取样数据，同时在具体调度资源的时候将其考虑在内。例如，2CPU单位的物理服务器X（慢机器）等同于0.8 CPU 的物理服务器Y（快机器）。根据这个信息，任务编排系统可以根据某个进程所处机器的类型调节CPU预设值。为了将这个任务变得简单，我们创造了一个虚拟CPU单位（Google计算单元GCU）。我们用GCU来给CPU性

能建模，同时维护一个GCU与不同CPU类型的性能映射表。

无法预知的性能因素

简单轮询策略最大的难点可能在于物理服务器—或者更准确地说，后端任务的性能——由于几个无法预知的因素区别很大，无法用统计学来衡量。

两种较为常见的无法预知的性能因素如下所示。

坏邻居

物理服务器上的其他进程（经常是完全无关的，由其他团队运行的）可能会显著影响到我们进程的性能。我们观察到性能的波动可以达到20%。差异主要来源于对共享资源的竞争，例如内存缓存、带宽等很不明显的地方。例如，如果一个后端任务对外请求的延迟升高（由于网络带宽的竞争），同时执行的请求数量就会增加，可能会导致GC（garbage collection）数量的增多。

任务重启

当一个任务重启时，在数分钟内它经常需要更多的资源。举一个例子，我们发现这种情况更多地影响到Java平台，因为它需要动态优化代码。为了解决这个问题，我们为一些服务器增加了一些逻辑，使得该服务器先保持在跛脚鸭状态，同时预热（触发这些动态优化）一段时间，直到性能稳定为止。考虑到我们每天都要更新很多服务器（例如，发布新版本），这种任务重启带来的性能问题就很严重了。

如果你的负载均衡策略不能很好地处理这些无法预知的性能问题，那么就会在处理大规模部署的时候造成负载不均衡的后果。

最闲轮询策略

简单轮询策略的一个替代方案是让每个客户端跟踪子集中每个后端任务的活跃请求数量，然后在活跃请求数量最小的任务中进行轮询。

例如，假设一个客户端使用后端子集t0到t9，而目前每个后端的活

跃请求数量如下所示：

t0	t1	t2	t3	t4	t5	t6	t7	t8	t9
2	1	0	0	1	0	2	0	0	1

那么，在处理新请求时，客户端会将可用后端列表按最少连接数过滤（t2、t3、t5、t7、t8）并从中选择一个。假设选择了t2，那么现在连接状态表变成了：

t0	t1	t2	t3	t4	t5	t6	t7	t8	t9
2	1	1	0	1	0	2	0	0	1

假设现在还没有任何一个请求成功，在下一个请求时，可选后端变成了t3、t5、t7、t8。

现在假设我们已经发送了4个新请求。仍然没有任何请求在这之间完成，那么目前的连接状态表变成了：

t0	t1	t2	t3	t4	t5	t6	t7	t8	t9
2	1	1	1	1	1	2	1	1	1

这时，可选后端变成了除了t0和t6之外的所有任务。然而，如果任务t4的请求结束了，那么当前状态则变成了“0　活跃请求”，那么新请求就会发送给t4。

这个算法的实现中实际采用了轮询，不过仅仅是在活跃请求数最小的任务里进行。如果不做这层过滤，那么这个策略就不能将负载分担得很好。最闲策略的思路来源于负载高的任务一般比负载低的任务延迟高，这个策略可以很好地将负载从这些负载高的任务迁走。

但是尽管这样，我们还是踩到了最闲轮询策略的最危险的一个坑：如果一个任务目前不健康，它可能会开始返回100%的错误。取决于错误的类型，错误回复可能延迟非常低；一般来说返回一个“我不健康”的错误比实际处理请求要快得多。于是，客户端任务错误地认为该任务可用，从而给该异常任务分配了大量的请求。我们在这里将这种问题称之为地漏效应（sinkhole effect）。幸运的是，这个问题有一个相对简单的解决办法——将最近收到的错误信息计算为活跃请求。这样，如果某个后

端进入异常状态，负载均衡策略可以开始将负载迁走，正如迁走过载项目的负载那样。

最闲轮询策略有两个很重要的限制：

活跃请求的数量并不一定是后端容量的代表

很多请求都需要花费相当长的时间等待网络请求（即等待它们触发的请求的回复），而非常少的时间在实际运行。例如，某个后端任务相对另外一个可以同时处理两倍的请求（因为它运行在一个有两倍性能的CPU服务器上），但是这些请求的延迟与其他任务基本一致（因为任务大部分时间都在等待网络）。在这个例子中，等待I/O消耗0 CPU，非常少的内存，以及0带宽。我们当然希望给这个更快的后端发送两倍的请求，但是最闲轮询策略会认为所有任务都是同样的负载。

每个客户端的活跃请求不包括其他客户端发往同一个后端的请求

每个客户端对每个后端任务状态仅仅有一个有限的视图：该客户端自己发送的请求。

在实际中，我们发现使用最闲轮询策略的大型服务会造成最忙任务比最闲服务使用两倍的CPU。该策略与简单轮询策略一样，效果都很差。

加权轮询策略

加权轮询策略通过在决策过程中加入后端提供的信息，是一个针对简单轮询策略和最闲轮询策略的加强。

加权轮询策略理论上很简单：每个客户端为子集中的每个后端任务保持一个“能力”值。请求仍以轮询方式分发，但是客户端会按能力值权重比例调节。在收到每个回复之后（包括健康检查的回复），后端会在回复中提供当前观察到的请求速率、每秒错误值，以及目前资源利用率（一般来说，是CPU使用率）。客户端根据目前成功请求的数量，以及对应的资源利用率成本进行周期性调节，以选择更好的后端任务处理。失败的请求也会记录在内，对未来的决策造成影响。

在实践中，加权轮询策略效果非常好，极大降低了最高和最低负载任务的差距。图20-6显示了一个后端任务子集从最闲轮询策略切换到加权轮询策略时的CPU变化。最高和最低负载任务的差距极大地降低了。

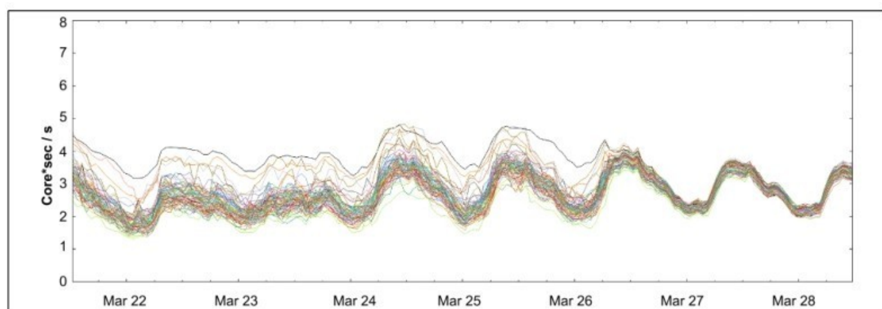


图20-6：在启用加权轮询策略前后的CPU分布图

第21章 应对过载

作者：**Alejandro Forero Cuervo**

编辑：**Sarah Chavis**

避免过载，是负载均衡策略的一个重要目标。但是无论你的负载均衡策略效率有多高，随着压力不断上升，系统的某个部位总会过载。运维一个可靠系统的一个根本要求，就是能够优雅地处理过载情况。

应对过载的一个选项是服务降级：返回一个精确度降低的回复，或者省略回复中一些需要大量计算的数据。例如：

- 平时在整个文档库中进行搜索，以针对某个查询返回最佳结果。而过载情况下仅仅在文档库的一小部分中进行搜索。

- 使用一份本地缓存的、可能不是最新的数据来回复，而不使用主存储系统。

然而，在极端的过载情况下，该服务甚至可能连这种降级回复都无法生成和发送。在这种情况下，该服务可能除了返回错误之外没有什么其他的好办法。一种避免这种场景发生的方法是，通过在数据中心之间

调度流量，做到每个数据中心都有足够的容量来处理请求。例如，如果一个数据中心运行了100个后端任务，其中每个后端任务可以处理500 QPS的请求，那么负载均衡策略就不会调度超过50,000 QPS的用户流量到这个数据中心。但是，在大规模部署时，这样的策略可能还是不够的。无论如何，构建能良好处理资源限制的客户端和对应的后端任务是最好的：在可能的情况下重定向请求，在必要时返回降级回复，同时在最差情况下，能够妥善地处理资源受限导致的错误。

QPS陷阱

不同的请求可能需要数量迥异的资源来处理。某个请求的成本可能由各种各样的因素决定，例如客户端代码的不同（有的服务有很多种客户端实现），或者甚至是当时的现实时间（家庭用户和企业用户，交互型请求和批量请求）。

Google在多年的经验积累中得出：按照QPS来规划服务容量，或者是按照某种静态属性（认为其能指代处理所消耗的资源：例如某个请求所需要读取的键值数量）一般是错误的选择。就算这个指标在某一个时间段内看起来工作还算良好，早晚也会发生变化。有些变动是逐渐发生的，有些则是非常突然的（例如某个软件的新版本突然使得某些请求消耗的资源大幅减少）。这种不断变动的目标，使得设计和实现良好的负载均衡策略使用起来非常困难。

更好的解决方案是直接以可用资源来衡量可用容量。例如，某服务可能在某个数据中心内预留了500 CPU 内核和1TB内存用以提供服务。用这些数字来建模该数据中心的服务容量是非常合理的。我们经常将某个请求的“成本”定义为该请求在正常情况下所消耗的CPU时间（这里要考虑到不同CPU类型的性能差异问题）。

在绝大部分情况下（当然总会有例外情况），我们发现简单地使用CPU数量作为资源配给的主要信号就可以工作得很好，原因如下：

- 在有垃圾回收（GC）机制的编程环境里，内存的压力通常自然而然地变成CPU的压力（在内存受限的情况下，GC会增加）。

- 在其他编程环境里，其他资源一般可以通过某种比例进行配置，以便使这些资源的短缺情况非常罕见。

如果过量分配其他非CPU资源不可行的话，我们可以在计算资源消耗的时候将各种系统资源分别考虑在内。

给每个用户设置限制

过载应对策略设计的一个部分是决定如何处理全局过载（global overload）的情况。在理想情况下，每个团队都能和他们所依赖的后端服务团队之间协调功能发布，从而使后端服务永远有足够容量服务最终用户，这样全局过载情况就永远不会发生。不幸的是，现实总是残酷的。全局过载情况在实际运行中出现得非常频繁（尤其是那些被很多其他团队使用的内部服务）。

当全局过载情况真的发生时，使服务只针对某些“异常”客户返回错误是非常关键的，这样其他用户则不会受影响。为了达到这个目的，该服务的运维团队和客户团队协商一个合理的使用约定，同时使用这个约定来配置用户配额，并且配置相应的资源。

例如，如果一个后端服务在全世界范围内分配了10,000个CPU（分布在多个数据中心中），它们的每用户限额可能与下面的类似：

- 邮件服务（Gmail）允许使用4,000 CPU（每秒使用4,000个CPU）。
- 日历服务（Calendar）允许使用4,000 CPU
- 安卓服务（Android）允许使用3,000 CPU
- Google+允许使用 2,000 CPU
- 其他用户允许使用 500 CPU

这里要注意的是，上述这些数字的总和会超过该后端服务总共分配的10000 CPU容量，这是因为所有用户都同时将他们的资源配额用满是一种非常罕见的情况。

我们随后从所有的后端任务中实时获取用量信息，并且使用这些数据将配额调整信息推送给每个后端任务。该系统的实现细节，已经超出了本书讨论的范围，但是可以说的是，我们在后端任务中写了相当多的代码来做到这一点。这里比较有趣的一个技术难点是实时计算每个请求所消耗的资源（尤其是CPU）。这种计算对某些不是按 “每个请求一个线程”模式设计的软件服务器尤其困难，这种软件用非阻塞API和线程池模式来处理每个请求的不同阶段。

客户端侧的节流机制

当某个用户超过资源配额时，后端任务应该迅速拒绝该请求，返回一个“用户配额不足”类型的错误，该回复应该比真正处理该请求所消耗的资源少得多。然而，这种逻辑其实不适用于所有请求。例如，拒绝一个执行简单内存查询的请求可能跟实际执行该请求消耗内存差不多（因为这里主要的消耗是在应用层协议解析中，结果的产生部分很简单）。就算在某些情况下，拒绝请求可以节省大量资源，发送这些拒绝回复仍然会消耗一定数量的资源。如果拒绝回复的数量也很多，这些资源消耗可能也十分可观。在这种情况下，有可能该后端在忙着不停地发送拒绝回复时一样会进入过载状态。

客户端侧的节流机制可以解决这个问题[71]。当某个客户端检测到最近的请求错误中的一大部分都是由于“配额不足”错误导致时，该客户端开始自行限制请求速度，限制它自己生成请求的数量。超过这个请求数量限制的请求直接在本地回复失败，而不会真正发到网络层。

我们使用一种称为自适应节流的技术来实现客户端节流。具体地说，每个客户端记录过去两分钟内的以下信息：

请求数量（requests）

应用层代码发出的所有请求的数量总计（指运行于自适应节流系统之上的应用代码）。

请求接受数量（accepts）

后端任务接受的请求数量。

在常规情况下，这两个值是相等的。随着后端任务开始拒绝请求，请求接受数量开始比请求数量小了。客户端可以继续发送请求直到requests＝K　*　accepts，一旦超过这个限制，客户端开始自行节流，新的请求会在本地直接以一定概率被拒绝（在客户端内部），概率使用公式21-1进行计算：

公式**21-1**：客户端请求拒绝概率

$$\max\left(0, \frac{\text{requests} - K \times \text{accepts}}{\text{requests} + 1}\right)$$

当客户端开始自己拒绝请求时，requests会持续上升，而继续超过accepts。这里虽然看起来有点反直觉，因为本地拒绝的请求实际没有到达后端，但这恰恰是这个算法的重点。随着客户端发送请求的速度加快（相对后端接受请求的速度来说），我们希望提高本地丢弃请求的概率。

我们发现自适应节流算法在实际中效果良好，可以整体上保持一个非常稳定的请求速率。即使在超大型的过载情况下，后端服务基本上可以保持50%的处理率。这个方式的一大优势是客户端完全依靠本地信息来做出决定，同时实现算法相对简单：不增加额外的依赖，也不会影响延迟。

对那些处理请求消耗的资源和拒绝请求的资源相差无几的系统来说，允许用50%的资源来发送拒绝请求可能是不合理的。在这种情况下，解决方案很简单：通过修改客户端中算法的accepts的倍值K（例如，2）就可解决：

- 降低该倍值会使自适应节流算法更加激进。
- 增加该倍值会使该算法变得不再那么激进。

举例来说，假设将客户端请求的上限从request＝2　*　accepts调整为request＝1.1*　accepts，那么就意味着每10个后端请求之中只有1个会被拒绝。

一般来说推荐采用K＝2，通过允许后端接收到比期望值更多的请求，浪费了一定数量的后端资源，但是却加快了后端状态到客户端的传递速度。举例来说，后端停止拒绝该客户端的请求之后，所有客户端检测到这个变化的耗时就会减小。

另外一个考量是，客户端节流可能不适用于那些请求频率很低的客户端。在这种情况下，客户端对后端状态的记录非常有限，任何想提高状态可见度的手段相对来说成本都较高。

重要性

重要性（criticality）是另外一个在全局配额和限制机制中比较有用的信息。某个发往后端的请求都会被标记为以下4类中的一种，这说明了请求的重要性。

最重要 CRITICAL_PLUS

为最重要的请求预留的类型，拒绝这些请求会造成非常严重的用户可见的问题。

重要 CRITICAL

生产任务发出的默认请求类型。拒绝这些请求也会造成用户可见的问题，但是可能没有CRITICAL_PLUS那么严重。我们要求服务必须为所有的CRITICAL和CRTICAL_PLUS流量配置相应的资源。

可丢弃的SHEDDABLE_PLUS

这些流量可以容忍某种程度的不可用性。这是批量任务发出的请求的默认值。这些请求通常可以过几分钟，或者几小时之后重试。

可丢弃的SHEDDABLE

这些流量可能会经常遇到部分不可用情况，偶尔会完全不可用。

我们发现以上4种分类可以描述大部分服务。我们曾经数次讨论过在中间增加更多的分类，以便更细粒度地描述请求。但是增加额外的值

需要增加处理这些信息的系统所需资源。

我们将重要性属性当成RPC系统的一级属性，花费了很多工夫将它集成进我们的很多控制手段中，以便这些系统在处理过载情况下可以使用该信息。例如：

- 当某个客户全局配额不够时，后端任务将会按请求优先级顺序分级拒绝请求（实际上，全局配额系统是可以按重要性分别设置的）。
- 当某个任务开始进入过载状态时，低优先级的请求会先被拒绝。
- 自适应节流系统也会根据每个优先级分别计数。

请求的优先级和该请求的延时性要求，也就是底层的网络服务质量（QoS）信息是不相关的。例如，当系统在用户输入搜索请求词语过程中实时显示搜索结果或者建议时，这些搜索请求的可丢弃性非常强（如果系统处于过载状态，这些结果也可以不显示），但是这些请求的延迟性仍然要求很高。

我们同时增强了RPC系统，可以自动传递重要性信息。如果后端接收到请求A，在处理过程中发出了请求B 和C给其他后端，请求B和C会使用与A相同的重要性属性。

在过去一段时间内，Google内部的许多系统都逐渐产生了一种与重要性类似的属性，但是通常不能跨服务兼容。通过标准化和在RPC系统中自动传递，我们现在可以在某些特定节点处统一设置重要性属性。这意味着，我们相信依赖的服务在过载情况下可以按正确的优先级来拒绝请求，不论它们处于整个处理栈中多深的位置。于是我们一般在离浏览器或者客户端最近的地方设置优先级—通常在HTTP前端服务器上。同时，我们可以在特殊情况下在处理栈的某处覆盖优先级设置。

资源利用率信号

我们的任务过载保护是基于资源利用率（utilization）实现的。在多数情况下，资源利用率仅仅是指目前CPU的消耗程度（目前CPU使用量除以全部预留CPU数量）。但是在某些情况下，同时也会考虑内存的

使用率。随着资源利用率的上升，我们开始根据请求的重要性来拒绝一些请求（高重要性的请求对应高阈值）。

我们使用的资源利用率信号是完全基于本地信息计算的（因为这个信号的作用就是为了保护任务自身），针对不同的信号有具体的实现。一个比较有用的信号是基于进程的“负载”，这是通过一个所谓的执行器负载均值（executor load average）决定的。

要计算执行器负载均值，我们要统计整个进程中的活跃线程数。在这里，“活跃”指那些正在运行，或者已经准备好运行，但是正在等待空闲CPU的线程。我们同时利用指数性衰变算法（exponential decay）来平滑这个值，当活跃线程数量超过该任务分配的CPU数量时开始拒绝请求。这意味着，当某个请求展开成非常大量的请求时（例如，某个请求被展开成突发性的大批短时请求），会导致负载值急剧上升，但是这个平滑过程基本上会将这个突发情况处理掉。但是如果这些请求不是短时的（负载值长时间保持在较高的水平），这些任务会开始拒绝请求。

虽然将执行器负载均值用在实践中被证实是一个非常有用的信号，但我们的系统可以接入后端自己定义的任意资源利用率信号。例如，我们可能会使用内存压力—表明了后端任务的内存使用率是否已经超出了正常运行范围—作为另一个可能的利用率信号。该系统还可以配置为同时使用多个信号，并且在超过综合（或者某个独立）目标利用率阈值的时候开始拒绝请求。

处理过载错误

在服务器端妥善处理过载请求之外，我们还仔细思考了在客户端接收到过载相关的错误信息时应该如何应对。在过载错误中，我们区分下列两种可能的情况：

数据中心中的大量后端任务都处于过载状态

如果跨数据中心负载均衡系统正在正常运行（意味着它可以传递状态，并且实时调度流量），这种情况应该不会出现。

数据中心中的一小部分后端任务处于过载状态

这种情况一般是由负载均衡系统的不完美造成的。例如，某个任务可能最近接收到了一个处理成本巨大的请求。在这种情况下，很有可能该数据中心仍然有其他容量可以处理该请求。

如果大部分后端任务都处于过载状态，请求应该不再重试，而应该一直向上传递给请求者（例如，向最终用户发送一个错误信息）。在更常见的、小部分任务过载的情况下，我们更倾向于立即重试该请求。一般来说，我们的跨数据中心负载均衡系统试图将客户端请求发往最近的数据中心。在特殊情况下，最近的数据中心仍然很远（如某个客户端可用的后端在另外一个大陆上），但是我们往往可以用其他手段解决这个问题。在这种情况下，重试某个请求造成的延时—一般是几个网络RTT—基本上是可以忽略的。

从负载均衡策略的角度看，重试请求和新请求是无法区分的。也就是说，我们并不会使用任何特殊的逻辑来保证某个重试请求真的会发往另外一个后端任务。在后端数量较多的情况下，这个“概率”本身就很大。增加新的逻辑确保请求发往不同的后端会将我们的API复杂度无谓地提高。

就算某个后端任务只是轻微过载，如果后端请求将重试和新请求同等对待，快速拒绝仍然是最好的选择。这些请求可以立刻在另外一个可能有空余资源的任务上重试。同等对待重试请求和新请求事实上形成了一种天然的负载均衡机制：可以将多余的负载自动转移给其他的任务。

决定何时重试

当某个客户端接收到“任务过载”错误时，需要决定是否要重试这个请求。我们针对大量后端任务过载的情况有几个方法来避免进行重试。

第一，我们增加了每次请求重试次数限制，限制重试3次。某个请求如果已经失败3次，我们会将该错误回应给调用者。这里的逻辑是指，如果一个请求已经三次选择了过载的任务，再重试也很可能无济于事，这时整个数据中心可能都处于过载状态。

第二，我们实现了一个每客户端的重试限制。每个客户端都跟踪重

试与请求的比例。一个请求在这个比例低于10%的时候才会重试。这里的逻辑是，如果仅仅只有一小部分任务处于过载状态，那么重试数量应该是相对较小的。

用一个实际例子来说（一个很糟糕的情况），我们假设一个数据中心正在接受一小部分请求，而大部分请求都被拒绝了。这里用X代表客户端逻辑向这个数据中心发出请求的速率。由于大部分请求会被重试，这里请求的数量将会增长得非常快，接近3X（由于每个请求会被重试3次）。虽然我们在这里限制了重试导致的请求数量，3倍的请求增长仍然是很多的，尤其是当拒绝请求的成本不能忽略的时候。然而，通过加入一个按客户端重试的限制（10% 重试比例），实际上我们将重试请求限制在大多数情况下仅增加为原来的1.1倍。这样的改进是很显著的。

第三个方法是客户端在请求元数据中加入一个重试计数。例如，这个计数器在第一次尝试时以0值开始，每次重试时加1，直到计数器为2时，请求重试限制会导致不再重试该请求。客户端会将近期信息记录为一个直方图。当某个后端需要拒绝请求时，它可以使用这个直方图信息来评判其他后端任务也处于过载的可能性。如果这些直方图信息表明大部分请求都有重试（意味着其他后端任务也可能处于过载状态），那么后端会直接返回“过载；无须重试”的错误，而不是标准的“任务过载”错误信息。

图21-1显示了在不同的情况下，每个后端接收到的请求的重试次数分布。这些数据是根据一个滑动窗口计算的（1000个初始请求，不计算重试）。为了更简单，这里客户端重试限制被忽略了（这些数据假设唯一的限制是每个请求重试3次），同时后端子集也可能会影响这些数据。

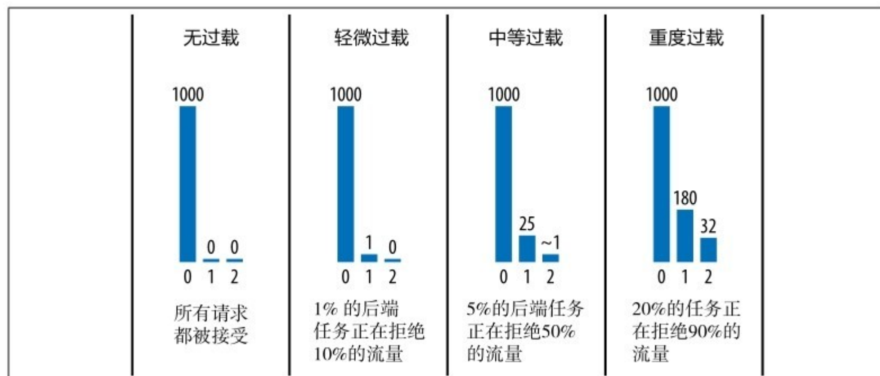


图21-1：不同情况下的请求直方图

Google的大型服务通常是由一个层次很深的系统栈组成的，这些系统可能互相依赖。在这种架构下，请求只应该在被拒绝的层面上面的那一层进行重试。当我们决定某个请求无法被处理，同时不应该被重试时，返回一个“过载；无须重试”错误，以避免触发一个大型的重试爆炸。

我们来看图21-2中所示的例子（实际上我们的系统栈要比这个复杂得多）。假设数据库前端（DB Frontend）目前正处于过载状态，拒绝请求。在这里：

- 后端任务B 会根据前述的规则重试请求。
- 然而当后端任务B确定数据库前端无法处理该请求时（例如，请求已经重试3次），后端任务B需要给后端任务A返回一个“过载；无须重试”错误，或者某个降级回复（假设它在无法连接数据库前端的情况下能产生某种可用的回复）。
- 后端任务A会根据它收到的回复进行处理。

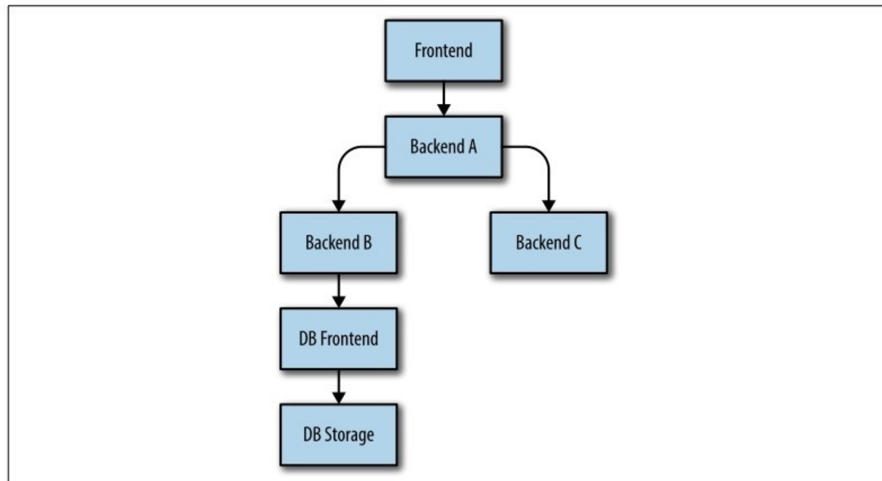


图21-2：某个依赖栈

这里的关键点是，数据库前端拒绝的请求应该仅仅在后端任务B处重试—在它的直接上层处。如果多层都要进行重试，会造成批量重试爆炸情况。

连接造成的负载

连接造成的负载是最后一个值得一提的因素。有时候我们仅仅考虑后端处理接收的请求所造成的负载（这也是用QPS来建模负载的一个问题），然而却忽略了其他因素，比如维护一个大型连接池的CPU和内存成本，或者是连接快速变动的成本。这样的问题在小型系统中可以忽略不计，但是在大型RPC系统中很快就会造成问题。

正如之前所说，我们的RPC协议需要不活跃的客户端定期执行健康检查。当某个连接空闲一段可配置的时间后，客户端放弃TCP连接，转为UDP健康检查。不幸的是，这种行为会对大量请求率很低的客户端造成问题：健康检查需要比实际处理请求更多的资源。通过仔细调节连接参数（如，大幅降低健康检查频率）或者动态创建和销毁连接可以优化这些场景。

处理突发性的新连接请求是另外一个（相关的）问题。我们观察到，超大规模的批处理任务会在短时间内建立大量的客户端。协商和维护这些超大数量的连接可以造成整个后端的过载。在我们的经验里，以

下策略可以帮助消除这些问题：

- 将负载传递给跨数据中心负载均衡算法（如，使用资源利用率进行负载均衡，而不仅仅是请求数量）。在这种情况下，过载请求会被转移到其他数据中心。

- 强制要求批处理任务使用某些特定的批处理代理后端任务，这些代理仅仅转发请求，同时将回复转发给客户端。于是，请求路线从“批处理客户端→后端”变为“批处理客户端→批处理代理→后端”。在这种情况下，当大型的批处理任务执行时，只有批处理代理任务会受到影响，而保护了真正的后端任务（也随之保护了其他的高优先级客户端）。这里，批处理代理任务实际充当了保险丝的角色。另外一个使用代理方式的优势是，我们可以减少后端连接的数量，同时提高负载均衡的效率（例如，代理任务可以使用更大的子集数量，同时可以更好地进行负载均衡）。

小结

本章和第20章讨论了如何利用不同的技术手段（确定性算法、加权轮询、客户端侧的节流、用户配额等）更平均地将负载分散到数据中心中。然而这些手段都依赖于在分布式下的状态传递机制。虽然大部分情况下都表现良好，但是在真实情况下，某些应用遇到了一些困难。

所以，我们认为保护某个具体任务，防止过载是非常重要的。简单地说：一个后端任务被配置为服务一定程度的流量，不管多少额外流量被指向这个任务，它都应该保证服务质量。并且由此得出，后端任务不应该在过载情况下崩溃。这些要求应该在一定流量范围内得到满足—有可能是两倍的配置流量，甚至10倍。我们可以接受超出某阈值时系统会崩溃的情况，但是应该将该阈值提升到某种很难发生的程度。

这里的关键是严肃对待降级情况。当这些降级情况被忽略时，很多系统都会显示出非常糟糕的表现。随着工作堆积，任务最终导致内存超标而崩溃（或者基本上花费所有CPU进行GC）、延迟上升、请求被忽略并且任务互相竞争资源。如果不解决这个问题，某个子系统的问题（如某一个后端的一个任务）可能会造成其他系统组件的失败，也有可能会

造成整个系统（或者是显著的大部分）失败。这种层级传递的失败是每个大型部署系统都必须考虑的关键点（更多知识参见第22章）。

一个常见的错误是认为过载后端应该拒绝和停止接受所有请求。然而，这个假设实际上是与可靠的负载均衡目标相违背的。我们实际上希望客户端可以尽可能地继续接受请求，然后在有可用资源时才处理。某个设计良好的后端程序，基于可靠的负载均衡策略的支持，应该仅仅接受它能处理的请求，而优雅地拒绝其他请求。

虽然我们有很多工具可以用来实现良好的负载均衡和过载保护机制，但是这里没有万能药：要进行负载均衡经常需要深入了解一个系统和它的请求处理语义。本章所描述的技术是根据Google内部很多系统的需求变化而产生的，可能会随着系统的变化而再次演进。

第22章 处理连锁故障

作者：**Mike Ulrich**

如果请求没有成功，以指数型延迟重试。

——Dan Sandler,Google软件工程师

为什么人们总是忘记增加一点点抖动因素呢？

——Ade Oshineye,Google 开发者布道师

连锁故障是由于正反馈循环（positive feedback）导致的不断扩大规模的故障。[72]连锁故障可能由于整个系统的一小部分出现故障而引发，进而导致系统其他部分也出现故障。例如，某个服务的一个实例由于过载出现故障，导致其他实例负载升高，从而导致这些实例像多米诺骨牌一样一个一个全部出现故障。

本章会通篇使用前文提到的莎士比亚搜索服务（见第2章）作为例子。该服务的生产环境配置如图22-1所示。

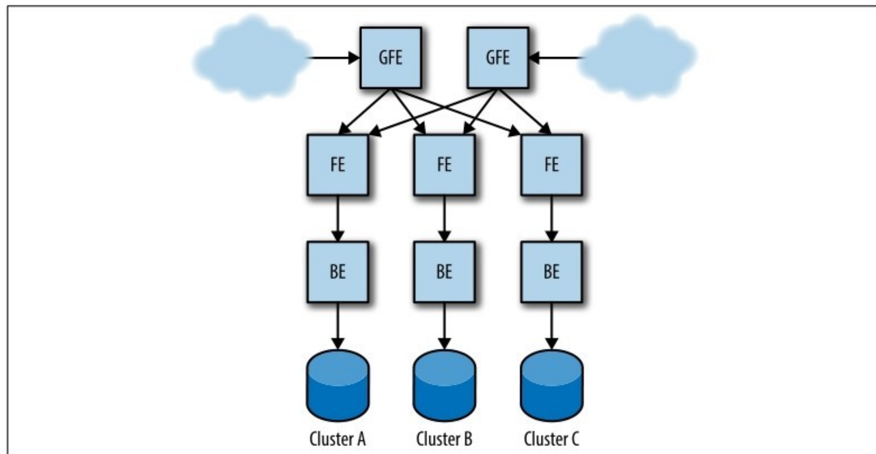


图22-1：莎士比亚搜索服务的生产环境配置图

连锁故障产生的原因和如何从设计上避免

一个设计良好的系统应该考虑到几个典型的连锁故障产生场景，从而在设计上避免它们发生。

服务器过载

最常见的连锁故障触发原因是服务器过载。这里讨论的多数连锁故障要么是直接由于服务器过载导致，要么是间接由于服务器过载引发的其他问题导致。

假设前端服务器在集群A中正在处理 1000 QPS的请求，如图22-2所示。

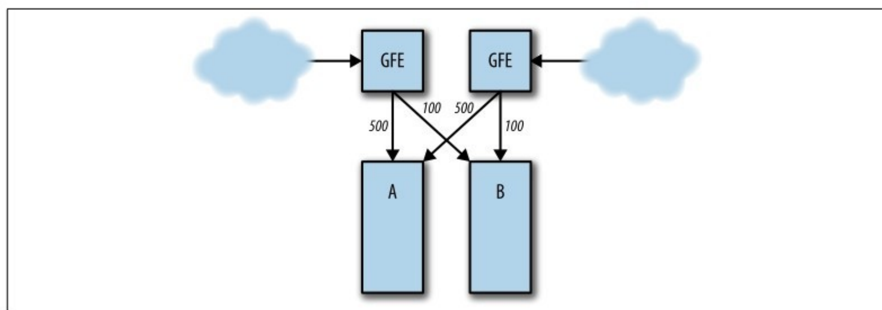


图22-2：正常的服务器负载在集群A/B之间的分布

如果集群B出现故障，如图22-3所示，发往集群A的请求上升至1200 QPS。集群A中的前端服务器无法处理这么多请求，由于资源不够等原因导致崩溃、超时，或者出现其他异常情况。结果，集群A成功处理的请求远低于之前的1000 QPS。

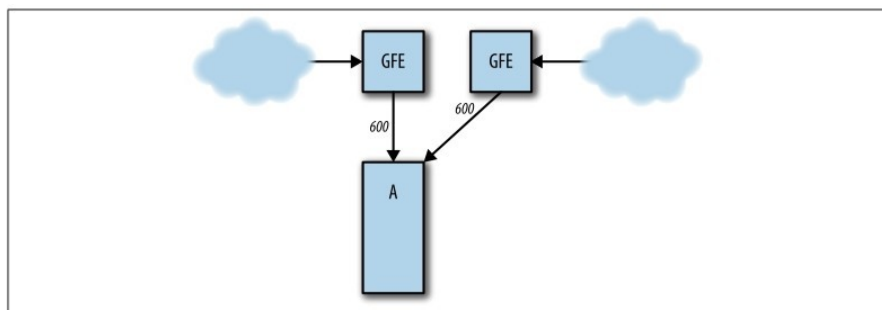


图 22-3：集群B出现故障，所有请求都发往集群A。

这种成功处理请求能力的下降可能会扩展到其他的集群中，甚至可能造成全球故障。例如，某个集群内部的过载可能会导致该服务实例崩溃；这时负载均衡器会将请求发送给其他集群，使那些集群的实例过载，从而造成整个服务过载故障。这些事件连锁发生的速度可能非常快（可能在分钟级范围），因为负载均衡器和任务编排系统的响应速度通常非常快。

资源耗尽

某一种资源的耗尽可以导致高延迟、高错误率，或者低质量回复的发生。这些的确是在资源耗尽时应该出现的情况：在负载不断上升到过载时，服务器不可能一直保持完全正常。

取决于究竟哪种资源最终耗尽，和软件服务器的构建方法，该情况可能会导致系统以低效率运行，甚至崩溃。负载均衡系统进而将请求转发给其他服务器，有可能导致整个集群请求处理成功率的下降，甚至使整个集群或者整个服务进入连锁故障模式。

不同种类的资源耗尽会对软件服务器产生不同的影响。

CPU

如果CPU资源不足以应对请求负载，一般来说所有的请求都会变慢。这个场景会造成一系列的副作用，包括如下几项。

正在处理的（in-flight）请求数量上升

因为处理请求需要较长的时间，同一时间服务器必须同时处理更多的请求（上升到一定数量可能会开始进入队列排队）。这会影响其他所有的资源，包括内存、活跃线程数（在每个请求一个线程的编程模型下）、文件描述符，和后端服务器的资源（该资源的耗尽可能会带来其他连锁问题）。

队列过长

如果没有足够的资源以稳定的速度处理所有请求，服务器会逐渐将请求队列填满。

这意味着延迟上升（因为所有请求都要排队一段时间），同时队列会使用更多的内存。

应对策略见本章后面“队列管理”小节。

线程卡住

如果一个线程由于等待某个锁而无法处理请求，可能服务器无法在合理的时间内处理健康检查请求（Borg系统会将这种情况视为服务器已经失败，从而杀掉它）。

CPU死锁或者请求卡住

服务器内置的看门狗机制（watchdog）[73]可能会检测到服务器无法进行工作，导致软件服务器最终由于CPU资源不够而崩溃。如果看门狗机制是远端触发的，但是由于请求队列排队，这些请求无法被及时处理，而触发看门狗机制杀掉进程。

RPC超时

服务器过载时，对客户端RPC的回复会变慢，最终会超过客户端所设置的超时时间。

这会导致服务器对请求实际进行的处理都被浪费了，而客户端可能会重试RPC，造成更严重的过载。

CPU缓存效率下降

CPU使用得越多，任务被分配到多个CPU核心上的几率越大，从而导致CPU核心本地缓存的失效，进而降低CPU处理的效率。

内存

就算没有其他副作用，同时处理的请求数量升高也会消耗更多的内存用于存放请求、回复以及RPC对象。内存耗尽可能导致如下情况的发

生。

任务崩溃

例如，某任务可能会因为超过资源限制而被容器管理器驱逐（VM或者其他的），或者程序自身逻辑会触发崩溃。

Java 垃圾回收（GC）速率加快，从而导致CPU使用率上升

一个糟糕透顶的场景：由于CPU资源减少，请求处理速度变慢，内存使用率上升，导致GC触发次数增多，导致CPU资源的进一步减少。我们将此称之为“GC死亡螺旋”。

缓存命中率下降

可用内存的减少可能会导致应用层缓存的命中率降低，导致向后端发送更多的RPC，可能会导致后端任务过载。

线程

线程不足可能会导致错误或者导致健康检查失败。如果服务器为此增加更多线程，这些线程可能会占用更多内存。在极端情况下，线程不足可能会导致进程ID数不足（Linux的进程ID数是有限的）。

文件描述符

文件描述符（file descriptor）不足可能会导致无法建立网络连接，进而导致健康检查失败。

资源之间的相互依赖

注意，很多资源的耗尽都会导致其他资源出现问题—某个服务过载经常会出现一系列次级现象看起来很像根本问题，这会使定位问题更困难。

假设如下场景：

1.某Java前端服务器GC参数没有被调优。

2.在高负载（但是在期待范围内）情况下，前端由于GC问题导致CPU不足。

3.CPU不足导致请求处理变慢。

4.同时处理的请求增多导致内存使用上升。

5.内存压力上升，同时由于固定内存分配比例的原因，用于缓存的内存数量减少。

6.缓存数量降低意味着缓存中键值数量下降，从而导致命中率下降。

7.缓存命中率下降导致更多的请求被发往后端进行处理。

8.后端服务器CPU或者线程不足。

9.CPU不足导致健康检查失败，从而触发了连锁故障。

在上述这个复杂情景下，发生故障时可能没有时间仔细分析因果关系。尤其是在前端和后端由不同团队运维时，判断后端崩溃是由于前端缓存命中率下降可能非常困难。

服务不可用

资源耗尽可能导致软件服务器崩溃。例如，这些服务器可能会由于内存超标而崩溃。一旦几个软件服务器由于过载而崩溃，其他软件服务器的负载可能会上升，从而使它们也崩溃。这种问题经常如同滚雪球一样越来越严重，不多久全部服务器就会进入崩溃循环。这种场景经常很难恢复，因为一旦某个软件服务器恢复正常，它就会接收到大量请求的轰炸，几乎立即再次崩溃。

例如，如果某个服务在10,000 QPS的水平下正常服务，但是当11,000 QPS的时候进入连锁故障模式，降低负载到9,000 QPS通常也无法恢复。这是因为这时该服务仍然处于容量不足的状态；通常仅仅只有一小部分的软件服务器可以正常处理请求。正常的软件服务器数量通常取决于：系统重启任务的速度，该任务进入正常工作的时间和新启动的

任务能够承受过载请求的时间。在这个例子里，如果10% 的任务目前可以正常处理请求，那么请求速率必须降低到 1,000 QPS 才能使整个系统恢复稳定。

同样的，这些软件服务器可能对负载均衡层来说是处于不健康状态，从而导致负载均衡可用容量的降低：软件服务器可能进入跛脚鸭状态或者无法处理健康检查。这种情况和软件服务器崩溃很类似，越来越多的软件服务器呈现不健康状态，每个仍健康的软件服务器都在很短的一段时间内接受了大量请求而进入不健康状态，导致能够处理请求的软件服务器越来越少。

会自动避免产生错误的软件服务器的负载均衡策略会将这个问题加剧—某几个后端任务产生了错误，会导致负载均衡器不再向它们发送请求，进而使得其余软件服务器的负载上升，从而再次触发滚雪球效应。

防止软件服务器过载

下面描述了避免过载的几种策略，大致以优先级排序。

使用负载压力测试得出服务器的极限，同时测试过载情况下的失败模式

为了避免过载，这一步是最重要的。除非在真实环境下测试，否则很难预测哪种资源会耗尽，以及资源耗尽产生的效果。细节请见本章后面的“连锁故障的测试”一节。

提供降级结果

给用户返回低质量的，但是更容易计算的结果。这里的策略取决于每个服务自己，细节见本章后面的“流量抛弃和优雅降级”一节。

在过载情况下主动拒绝请求

软件服务器应该保护自己不进入过载崩溃状态。当前端或者后端进入过载模式时，应尽早尽快地将该请求标记为失败。细节见本章后面的“负载抛弃和优雅降级”一节。

上层系统应该主动拒绝请求

注意，由于速率限制实现中一般不会考虑服务整体健康程度的因素，它可能并不能阻止一个已经发生了的连锁故障。简单的速率限制实现很有可能会导致一些容量的浪费。速率限制可以具体在以下位置实现：

- 在反向代理层，通过针对请求的某种特性进行数量限制（如IP地址），来缓解和避免拒绝服务攻击，避免攻击性客户端的影响。

- 在负载均衡器层，在服务进入全局过载时主动丢弃请求。取决于该服务的实现和复杂度，这种丢弃可能是针对所有请求的（丢弃超过X QPS 的所有请求），或者是有选择性的（丢弃非最近用户发来的请求，或者丢弃低优先级请求，如背景同步请求，但是保留用户交互型请求）。

- 在每个任务自身，避免负载均衡层的随机扰动导致软件服务器过载。

进行容量规划

好的容量规划可以降低连锁反应发生的可能性。容量规划应该伴随着性能测试进行，以确定可能导致服务失败的负载程度。例如，如果每个集群的临界点是5,000 QPS，服务负载平均分布，[74]而该服务的峰值负载是19,000 QPS，我们则需要大约6个集群来以N＋2模式运行该服务。

进行容量规划只能减少触发连锁反应的可能性，但是并不能完全避免。当一个计划内或者计划外的事件导致大部分集群容量同时下线时，连锁反应是不可避免的。负载均衡问题、网络分区事件，或者突发性流量增长，都会创造意料之外的负载问题。有些系统可以根据需要动态增加容量，这可能防止过载发生，但是适当地进行容量规划还是必要的。

队列管理

大部分“每个请求一个线程”模型的软件服务器使用一个队列和一个

线程池来处理请求。接收到的请求将会进入队列，线程池中的某线程会将请求从队列中取出，进行实际处理。通常情况下，当队列排满时，服务器会拒绝新的请求。

如果请求速率和单个请求处理耗时是固定的，那么排队就没有必要：同时运行的线程数量是固定的。在这个理想化的情景下，只有在请求速率超过单个请求的处理速率时，请求才会进入队列，这种情况会导致线程池和队列的同时饱和。

在队列中的排队请求消耗内存，同时使延迟升高。例如，如果队列大小是线程数量的10倍，而单个线程处理单个请求的耗时是100ms。如果队列处于满载状态，每个请求都需要1.1s才能处理完成，大部分时间都消耗在排队过程中。

对一个流量基本稳定的服务来说，队列长度比线程池大小更小会更好（如 50% 或更小）。当服务处理速度无法跟上请求到达速率时，尽早拒绝请求会更好。例如，Gmail 通常使用无队列软件服务器，在线程满的时候转移负载到其他服务器上。而另外一些经常有“突发性”负载的系统，或者请求模式经常变动的软件服务器通常基于线程数量、请求处理时间和突发性流量的频率和大小来计算队列长度。

流量抛弃和优雅降级

流量抛弃（load shedding）是指在软件服务器临近过载时，主动抛弃一定量的负载。目标是避免该软件服务器出现内存超限，健康检查失败，延迟大幅升高，或者其他过载造成的现象。也就是使软件服务器在负载极限时，尽可能地再多做一些有用的工作。

一种简单的流量抛弃实现方式是根据CPU使用量、内存使用量及队列长度等进行节流。限制队列长度的详细讨论请参见上一小节“队列管理”。例如，一个有效的办法是当同时处理的请求超过一定量时，开始直接针对新请求返回 HTTP 503（服务不可用）。

另外的做法包括将标准的先入先出（FIFO）队列模式改成后入先出（LIFO），以及使用可控延迟算法（CodDel）（参见文献[Nic12]），

或者类似的方式更进一步地避免处理那些不值得处理的请求（参见文献[Mau15]）。如果某个用户的Web搜索请求由于RPC排队因素已经等待了10s，很有可能该用户已经放弃了，已经刷新了浏览器又发送了一个新请求：这时回复第一个RPC请求已经没有任何意义，因为它已经没用了！这个策略和层层向下传递RPC截止时间的策略结合起来，工作得十分好，可参见本章后面的“请求延迟和截止时间”一节。

其他更复杂的手段包括通过精确识别客户端的方式来更有选择地丢弃部分任务，或者将请求按优先级排序，按优先级处理等。这些策略更适用于一些基础的共享服务采用。

优雅降级（graceful degradation）可在流量抛弃的基础上进一步减少服务器的工作量。在某些应用程序中，是可以通过降低回复的质量来大幅减少所需的计算量或者所需的计算时间的。例如，一个搜索类型的应用可能在过载情况下仅仅搜索保存在内存中的数据，而不是搜索全部存在硬盘上的数据。该服务或者可以采用一种不那么精确（但是更快）的算法来进行结果排序。

当我们评估流量抛弃或者优雅降级时，需要考虑以下几点：

- 确定具体采用哪个指标作为流量评估和优雅降级的决定性指标（如，CPU用量、延迟、队列长度、线程数量、是否该服务可以自动进行降级，或者需要人工干预）。

- 当服务进入降级模式时，需要执行什么动作？

- 流量抛弃或者优雅降级应该在服务的哪一层实现？是否需要在整个服务的每一层都实现，还是可以选择某个高层面的关键节点来实现？

当评估和具体实施的时候，还需要考虑以下几点：

- 优雅降级不应该经常被触发—通常触发条件显示了容量规划的失误，或者是出现了意料之外的负载转移。整个降级系统应该简单、易懂，尤其是在不经常使用的情况下。

- 记住，代码中平时不会使用的代码路径（通常来说）是不工作

的。在稳定运行状态下，优雅降级不会经常触发，意味着在这个模式下的运维经验很少，对这个模式的问题也不够熟悉，这就会升高这种模式的危险性。我们可以通过定期针对一小部分的服务压力测试以便更多地触发这个模式，保证这个模式还能正常工作。

- 监控系统应该在进入这种模式的软件服务器过多时报警。

- 复杂的流量抛弃和优雅降级系统本身就可能造成问题—过于复杂的逻辑可能会导致软件服务器以外的服务进入降级模式运行，甚至进入反馈循环。设计时应该实现一种简单的关闭降级模式，或者是快速调节参数的方式。将这个配置文件存储在一个强一致性的存储系统（如Chubby）中，每个软件服务器都可以订阅改变，可以提高部署速度，但是同时也会增加整个系统的同步性风险（如果配置文件有问题，全部服务器同时都会受到影响）。

重试

假设前端中的代码与后端通信时，已经实现了重试机制。该重试机制在遇到错误时重试，同时限制每个逻辑请求的RPC数量为10。如下面这段代码，使用Go语言与gRPC实现：

```
func exampleRpcCall(client pb.ExampleClient, request pb.Request) *pb.Response {

    // 将 RPC 超时设置为 5s
    opts := grpc.WithTimeout(5 * time.Second)

    // 最多重试 RPC 20 次
    attempts := 20
    for attempts > 0 {
        conn, err := grpc.Dial(*serverAddr, opts...)
        if err != nil {
```

```
            // 出现了问题，重试
            attempts--
            continue
        }
        defer conn.Close()

        // 创建客户端，发送 RPC
        client := pb.NewBackendClient(conn)
        response, err := client.MakeRequest(context.Background, request)
        if err != nil {
            // 发送请求过程中出现问题，重试
            attempts--
            continue
        }

        return response
    }

    grpclog.Fatalf( "ran out of attempts" )
}
```

这个系统可能会造成如下的连锁故障：

1.假设后端服务每个任务的负载上限是10,000 QPS，在超过这个上限之后，优雅降级系统会拒绝所有的额外请求。

2.前端代码以固定的 10,100 QPS 调用MakeRequest，使得后端过载100 QPS，后端会将这些请求拒绝掉。

3.这些100 QPS的失败请求会在MakeRequest中以1,000ms为周期重试，这些请求可能会成功。但是这些请求本身叠加上正常的请求，后端现在会接收到 10,200 QPS —200 QPS 的请求会因为过载而失败。

4.重试的量在增加：第一秒的100 QPS，造成了第二秒的 200 QPS，造成了300 QPS。越来越少的请求能够一次成功，也就是说，后端实际进行的有价值工作越来越少。

5.如果后端任务无法处理连续增长的负载—这些请求会消耗文件描述符、内存、CPU时间等—后端任务可能会在请求压力和重试下最终崩溃。这个崩溃会将压力重新分配到其他的后端任务上，进而造成这些任务的过载。

这里为了描述这个场景，简化了一些细节，[75]但是这里很好地展

现了重试是如何摧毁一个系统的。注意临时性的过载升高，或者使用量的缓慢增加都有可能造成这种情况。

就算MakeRequest的请求速率降低到系统崩溃前的级别（如9,000 QPS），还取决于后端返回错误的成本大小，这个问题很可能不会消失。这里主要有两个因素：

- 如果后端使用大量的资源处理最终这些会失败的请求，那么重试造成的请求可能仍然会将后端保持在过载状态中。

- 后端服务器自身可能也不稳定。重试可以放大本章前面“服务器过载”一节中提到的效应。

如果上述任意一个情况属实，为了脱离这种循环，必须大幅减少甚至彻底消除前端产生的负载，直到重试停止，后端稳定为止。

这个模式在多起连锁故障中出现，不管前端和后端是采用 RPC 消息模式通信，还是由客户端运行的 JavaScript 代码发送XmlHttpRequest调用，并且重试，又或者是来自于某种离线同步协议在错误情况下大量重试导致的。

当发送自动重试时，需要将如下部分考虑在内：

- 大部分后端保护策略都适用于此。尤其是，对系统进行测试可以提前显现问题，同时优雅降级可以将重试对后端的影响降低。

- 一定要使用随机化的、指数型递增的重试周期。可参见文献[Bro15]中提到的文章Exponential Backoff and Jitter（http://www.awsarchitectureblog.com/2015/03/backoff.html）。如果重试不是随机分布在重试窗口里的，那么系统出现的一个小故障（某个网络问题）就可能导致重试请求同时出现，这些请求可能会逐渐放大（参见文献[Flo94]）。

- 限制每个请求的重试次数。不要将请求无限重试。

- 考虑使用一个全局重试预算。例如，每个进程每分钟只允许重试

60次，如果重试预算耗尽，那么直接将这个请求标记为失败，而不真正发送它。这个策略可以在全局范围内限制住重试造成的影响，容量规划失败可能只是会造成某些请求被丢弃，而不会造成全球性的连锁故障。

- 从多个视角重新审视该服务，决定是否需要在某个级别上进行重试。这里尤其要避免同时在多个级别上重试导致的放大效应：高层的一个请求可能会造成各层重试次数的乘积数量的请求。如果服务器由于过载不能提供服务，后端、前端、JavaScript层各发送3次重试（总计4次请求），那么一个用户的动作可能会造成对数据库的64 次请求（4^3）。在数据库由于过载返回错误时，这种重试只会加重问题。

- 使用明确的返回代码，同时详细考虑每个错误模式应该如何处理。例如，将可重试错误和不可重试错误分开。不要在客户端代码里重试永久性错误或者异常请求（malformed request），因为这两种请求都永远不可能成功。当服务过载时，返回一个详细的信息，这样客户端和其他层可以加大延时，甚至不再重试。

在紧急情况下，可能很难确定某次事故是否是由于重试机制导致的。重试速率的图表可以帮助识别重试行为，但是也可能会被当成一个故障现象，而不是故障原因。在处理过程中，这可以被认为是容量不足的一个特殊情况。要注意的是，在这种情况下，我们要么需要修复重试行为（这经常意味着更改代码），要么需要大幅降低负载，要么需要彻底消除重试请求。

请求延迟和截止时间

当某个前端任务发送RPC给后端服务器时，前端需要消耗一定资源等待后端回复。RPC截止时间（deadline）定义了前端会等待多长时间，这限制了后端可以消耗的前端资源。

选择截止时间

设置一个截止时间通常是明智的。不设置截止时间，或者设置一个非常长的截止时间通常会导致某些短暂的、已经消失的问题继续消耗服务器资源，直到重启。

截止时间设置得太长可能会导致框架中的高层级部分由于低层级的问题而持续消耗资源。截止时间设置得太短可能会导致某些比较重型的请求持续失败。恰当的截止时间设置，需要在多个限制条件中选择一个平衡点。

超过截止时间

很多连锁故障场景下的一个常见问题是软件服务器正在消耗大量资源处理那些早已经超过客户端截止时间的请求。这样的结果是，服务器消耗大量资源没有做任何有价值的工作，回复已经被取消的RPC是没有意义的。

假设一个RPC请求设置了10s的截止时间，由客户端设置。服务器目前过载严重，导致从队列到线程池的时间需要11s。这时，客户端已经放弃了该请求。在多数情况下，服务器再继续处理这个请求是不明智的，因为这样无法产生任何价值—客户端已经放弃了这个请求，不会再接收回复。

如果处理请求的过程有多个阶段（例如，由一系列 RPC和回调函数组成），该软件服务器应该在每个阶段开始前检查截止时间，以避免做无用功。例如，如果一个请求被分解为解析请求阶段、发送后端请求阶段和处理阶段，那么可能在每个阶段开始之前都要检查是否还有足够剩余时间处理请求。

截止时间传递

与其在发送RPC给后端服务器时自拟一个截止时间，不如让软件服务器采用截止时间传递和取消传递的策略。

可使用截止时间传递机制，截止时间在整个服务栈的高层设置（如，前端服务器）。由初始请求触发的整个RPC树会设置同样的绝对截止时间。例如，如果服务器A选择使用30s截止时间，然后花费7s时间处理请求，再发送RPC给服务器B，那么从A发向B的RPC就会有23s的截止时间。如果服务器B处理请求花费4s，并且再向C发送请求，从B发向C的RPC就会有19s的截止时间，以此类推。理想情况下，在整个树状结构

里面的所有服务器都采用同样的截止时间传递机制。

如果不采用这种传递机制，那么下列场景就可能出现：

1.服务器A给服务器B发送了一个RPC，用10s截止时间。

2.服务器B等待了8s时间才开始处理请求，并且给服务器C发送了一个RPC。

3.如果服务器B使用截止时间传递机制，它应该设置2s的截止时间，但是这里假设它使用了一个在代码中写死的20s截止时间。

4.服务器C在5s后才将请求从队列中取出。

如果服务器B采用了截止时间传递机制，服务器C应该立刻放弃处理该请求，因为2s的截止时间已经过了。但是在上述场景中，服务器C会认为仍有15s的时间，从而继续处理该请求，但是这些工作都是无用功，因为从服务器A到服务器B的请求已经超出了截止时间。

同时，我们可能还会将传递出去的截止时间减少一点（如几百毫秒），以便将网络传输时间和客户端收到回复之后的处理时间考虑在内。

应该考虑给发出的截止时间设置一个上限，我们可能想要对非关键性后端的等待时间，或者那些通常非常快的RPC请求设置限制。然而，要确保先清晰了解服务流量的组成，否则可能会导致某种特定类型的请求永远失败（如，带大量数据的请求，或者那些需要大量计算的请求）。

这里有一些特例，某些服务器可能希望在截止时间过后继续处理该请求。例如，如果某个服务器在处理某个请求时需要执行某种很耗时的状态更新工作，同时周期性地将状态更新操作持久化，那么可能服务器应该仅仅在持久化操作之后才检查请求截止时间，而不是在耗时操作完成之后立刻检查。

RPC取消的传递可以避免某些泄露情况，如果某个初始RPC设置了

一个很长的截止时间，但是底层之间的RPC只有短暂的截止时间，超时失败了。使用简单的截止时间传递可能会导致初始RPC虽然无法继续处理，却继续消耗服务器资源直到超时。

请求延迟的双峰分布（**Bimodal**）

假设上面那个例子中的前端服务器由10个任务组成，每个任务有100个工作线程。这意味着前端服务器有共计1000个线程容量。在常规状态下，前端服务器发送1000QPS的请求，每个请求耗时100ms完成。这意味着前端服务器共计有100个工作线程同时工作（1000QPS * 0.1s）。

假设某个事件导致5%的请求永远不会结束。这可能是由于Bigtable某个行范围不可用，导致对这些部分的请求不可用。结果，5%的请求要一直到超时才能完成，而剩余的95%仍耗时100ms。

使用100s的截止时间，5%的请求会消耗5000个线程（50QPS * 100 seconds），但是前端服务器并没有这么多可用线程。忽略副作用，前端也仅能够处理19.6%的请求（1000可用线程 /（5000+95）线程工作），这会造成 80.4%的错误率。

因此，不仅 5% 的请求受到了影响（那些由于后端问题不可能成功的请求），实际上大部分的请求都受到了影响。

下列指导思想可以帮助解决这一类问题：

- 检测这个问题可能会很困难。尤其是当我们监控平均延迟的时候，很难发现原来是双峰分布导致的问题。当我们观测到延迟上升时，应该额外注意观察延迟的分布情况。

- 如果无法完成的请求能够尽早返回一个错误而不是等完整个截止时间，我们就可以避免这个问题。例如，如果一个后端服务器不可用，经常立刻返回一个错误值是最好的，而不是等待这个后端服务器变得可用。如果RPC层支持快速失败的选项，一定要启用它。

- 将截止时间设置得比平均延迟大好几个数量级通常是不好的。在

前述例子中，最开始只有一小部分请求超时，但是由于截止时间设置比平均延迟高三个数量级，导致了线程耗尽问题。

- 当使用按键值空间分布的某种共享资源时，应该考虑按键值分布限制请求数量，或者使用某种滥用跟踪系统。假设你的后端要处理来自不同客户端的性能和特征各异的请求，我们可以考虑限制一个客户端只能占用25% 的线程总数，以便在某个异常客户端大量产生负载的情况下提供一些公平性。

慢启动和冷缓存

进程在刚刚启动之后通常要比稳定状态下处理请求的速度慢一点。慢的原因可能是由下列一个或多个原因导致：

必需的初始化过程

在接收到第一个请求后，需要跟后端服务器建立连接。

运行时性能优化，尤其是Java

JIT 编译过程，热点优化，以及类延迟加载机制。

同样的，有些服务器会在缓存没有充满之前效率很低。例如，对某些Google的服务来说，大部分请求都是由缓存提供的，所以缓存没有命中的请求会非常慢。在稳定状态和一个热缓存下，只有很少请求没有命中缓存，但是当缓存是空的情况下，100%的请求都非常耗时。其他服务可能使用缓存将用户状态放置在内存中。这可以通过在反向代理和服务前端之间的硬黏性或软黏性（hard/soft stickiness）设置来达到。

如果某个服务不是按照100%冷缓存模式来配置部署的，那出问题的可能性会升高，同时应该采用一定步骤来避免问题的发生。

下列情况可能会导致冷缓存问题：

上线一个新的集群

刚刚增加的集群的缓存是空的。

在某个集群维护之后恢复服务状态

缓存中的数据可能是过期的。

重启

如果某个有缓存的任务最近重启了，那么它的缓存需要一定时间填充。可能有必要将缓存从一个软件服务器拿出到另外的独立服务器中，比如memcache，而且可以将缓存在多个服务器中共享，但是可能会多消耗一层RPC和额外的延迟问题。

如果缓存对服务造成了很大的影响，[76]可能要采取以下几种策略中的一种或多种：

- 过量配备（overprovision）该服务。区分延迟类缓存和容量类缓存是很重要的：当使用延迟类缓存时，服务器可以在空缓存的情况下仍然处理预期的请求负载，但是使用容量类缓存时，该服务将不能够在空缓存下处理请求负载。运维人员应该对增加缓存层非常警惕，应该确保每个新添加的缓存要么是延迟类缓存，要么是经过良好设计的、可安全使用的容量类缓存。有些时候加入缓存是为了提高性能，但是最后却变成了强制依赖。

- 使用通用的连锁故障避免手段。尤其是，服务器应该在进入过载状况，或者降级状况下主动拒绝请求，同时应该进行测试，以观察服务在大规模重启等情况下如何表现。

- 当为一个集群增加负载时，需要缓慢增加。初期的小流量会加热缓存，一旦缓存热起来，就可以增加更多的请求了。确保所有集群都处理一定程度的负载，以保证缓存随时是热的状态也是一个不错的选择。

保持调用栈永远向下

在前面的莎士比亚搜索服务的例子中，前端服务器和后端服务器通信，后端服务器随后和存储层通信。一个存储层的问题会造成和它通信的服务器的问题，修复存储层的问题通常会同时修复后端和前端服务器。

然而，假设后端服务器中的所有任务会彼此通信。例如，当存储层无法处理某个请求时，后端服务器可能会彼此之间代理请求。这种在层内的交互通信可能会导致问题，原因有如下几个：

- 这种通信容易导致分布式死锁。后端服务器可能使用同样的线程池来等待发送给其他后端的RPC，以及处理来自其他后端的请求。假设后端A的线程池满了，后端B给后端A发送了一个请求，于是占用了后端B的一个线程等待后端A的线程池可用。这种行为可能会导致线程耗尽问题的扩散。

- 如果这种交互通信是由于某种失败因素或者过载情况导致的（如负载重新分布机制在负载很高的时候很活跃），这种交互通信可能在延迟上升的时候从不常见变得很常见。

例如，假设一个用户有一个主后端和一个预先选择好的另外集群中的一个热备后端，主后端在底层出现错误或者延迟上升的情况下将请求代理给热备后端。如果整个系统都处于过载状态，那么从主到副的这种代理可能会增多，会给系统带来更多的负载，因为请求通常要被解析两次，还需要主后端消耗资源等待副后端任务。

- 取决于交互通信的重要性，初始化整个系统可能会变得更复杂。

在用户的请求路径中最好能够避免使用同层通信—也就是避免通信路径中出现环。应该由客户端来进行这种通信。例如，如果一个前端需要和后端通信，但是猜错了后端任务，后端不会代理请求给正确的后端，而是通过返回错误使得前端在正确的后端任务上重试它的请求。

连锁故障的触发条件

当某个服务容易产生连锁故障时，有一些情况可能触发多米诺骨牌效应。本节指出了某些触发条件。

进程崩溃

某些服务任务会崩溃，减少了服务可用容量。进程可能会由于接收到致死请求（query of death，会触发进程崩溃的RPC）而崩溃，或者由

于集群问题，代码中的断言错误，或者很多其他原因导致。一个非常小的事件（例如，几个任务崩溃，或者几个任务被转移到其他的物理机器上）都可能会导致某个服务进入崩溃边缘。

进程更新

发布一个新版本或者更新配置文件时，可能由于大量任务同时受影响而触发连锁故障。为了避免这种情况，必须要在设计进程更新机制的时候考虑到对容量的影响，或者在非峰值时间推送更新。根据请求数量和可用容量来动态调节任务的同时更新数量可能是个好办法。

新的发布

新的二进制文件、配置更改，或者底层架构的改变都可能导致请求特征的改变，资源使用和限制的改变、后端的改变和其他系统组件的改变可能导致连锁故障的发生。

在发生连锁故障时，检查最近的改变以及回滚通常是明智的，尤其在这些改变会影响容量或者更改请求特点的情况下。

你的服务通常应该实现某种改变记录，这样可以帮助尽快识别最新的改变。

自然增长

在很多情况下，连锁故障不是由于某个特定的服务改变导致的，而是由于使用量的天然上升，却没有进行对应的容量调整导致的。

计划中或计划外的不可用

如果服务是多集群部署的，某些容量可能会由于维护或者集群事故而不可用。同样的，某个服务的关键依赖可能不可用，导致上游服务的容量不可用，或者由于需要将请求发给另外一个集群而使延迟升高。

请求特征的变化

由于负载均衡配置的改动、用户流量的变化及集群问题等可能导致前端服务器负载的变化，这将导致后端服务可能会从另外一个集群接收

请求。同时，前端代码或者配置的改变可能会导致具体每个请求的平均成本发生变化。同样的，由于用户使用的增多或者行为的改变，该服务处理的数据可能已经变化了：例如，某个照片存储系统的每个用户照片数量和大小随着时间趋于上涨。

资源限制

某些集群操作系统允许超卖资源。CPU是一个模糊资源；通常机器会有一些备用CPU资源，以便在CPU峰值的时候提供一个安全防护。这种备用CPU资源在集群和集群机器之间都不一样。

依赖这些备用资源作为你的安全防护网是很危险的。这些备用资源的可用性完全取决于集群中其他任务的行为，所以可能随时消失。例如，如果一个团队启动了一个MapReduce任务，在很多机器上使用了很多CPU，那么总共的备用CPU资源就可能突然减少，导致无关任务的CPU资源不够。当进行压力测试时，应确保服务一直停留在预留的资源限制中，

连锁故障的测试

从理论上预测服务会以什么方式进入故障状态是很困难的。这一节讨论了检测服务是否受连锁故障影响的测试策略。

你应该针对服务进行压力测试，通过对重载下服务行为的观察可以确定该服务在负载很重的情况下是否会进入连锁故障模式。

测试直到出现故障，还要继续测试

理解服务在高负载情况下的行为模式可能是避免连锁反应最重要的一步。知道系统过载时如何表现可以帮助确定为了修复问题所需要完成的最重要的工程性任务。最不济这种知识也能够在紧急情况下帮助on-call工程师处理故障。

压力测试每个组件直到它们崩溃。随着压力上升，一个组件通常可以成功处理请求直到某一临界点不能再处理更多请求。在这个临界点上，组件理想情况下针对多余的负载返回错误或者降级的回复。但是不

应该显著降低它成功处理请求的速率。一个组件如果在过载情况下开始崩溃，或者返回大量错误信息就非常容易造成连锁故障。设计良好的组件应该可以拒绝一小部分请求而继续存活。

压力测试同时显示了临界点所在，这一点是容量规划流程的关键所在。这使得我们可以以此进行回归测试，按最差情况配备资源，或者使用利用率和安全系数做妥协。

因为缓存的影响，逐渐升高的负载可能与瞬间提升的负载性能差距很大。因此，应该考虑测试这两种场景。

我们还应该测试一个组件在过载之后再恢复到正常水平的行为状态。这种测试可能可以帮助回答下列问题：

- 如果一个组件在高负载条件下进入了降级模式，它是否能够在无人工干预的情况下退出该模式？

- 如果高负载情况下几个服务器崩溃，负载需要降低多少才能使系统重新稳定下来？

如果我们压力测试一个有状态的服务，或者使用缓存的服务，压力测试代码应该记录多次交互的状态，同时检查高负载情况下的回复正确性，这通常是某些非常难以查找的并行Bug出现的时候。

记住，每个独立组件都可能有不同的临界点，所以应该分别测试。我们不能够预先知道系统中的哪个组件先崩溃，但这恰恰是问题出现时最需要的信息。

如果你的系统有恰当的过载保护措施，那么可以考虑在生产环境中针对一小部分容量进行模拟故障测试，这样可以发现在真实流量情况下系统中的哪个组件先出问题。这种问题可能不能很好地由合成压力测试流量展示，所以真实流量的压力测试可能会提供更真实的结果，当然也可能会造成用户可见的问题。所以在测试真实流量的时候一定要小心：确保你有足够的可用额外容量以备自动保护措施失败，需要人工进行切换。可以考虑以下几种生产环境测试：

- 快速或者缓慢地降低任务数量，超越之前预期的流量模式。
- 快速去掉某一个集群的容量。
- 屏蔽不同的后端（试验超时等因素对系统的影响）。

测试最常用的客户端

理解最大的客户是如何使用服务的。例如，我们想知道客户端：

- 能够在服务中断的情况下排队。
- 遇到错误时使用随机化的指数型延迟进行重试。
- 是否会由于外部因素导致流量的突然变化（例如，某个外部软件更新可能会清空某个离线客户端的缓存）。

取决于服务本身，我们可能无法直接控制所有的客户端代码。但是，理解大客户是如何与服务交互的还是很有益处的。

同样的道理也适用于大型内部客户。针对最大的客户进行模拟灾难演习，以观察他们如何应对。询问内部客户是如何使用该系统的，以及他们是用什么手段来应对后端问题的。

测试非关键性后端

应该测试非关键性后端，以确保它们的不可用不会影响到系统中的其他关键性组件。

例如，如果某个前端同时有关键性和非关键性后端。通常情况下，一个请求同时需要关键性后端（如，搜索结果）和非关键性后端（如，拼写建议）。请求可能会受非关键性后端影响而变慢，或者占用更多资源。

在测试非关键性后端不可用之外，还应测试如果这些后端不返回结果前端如何表现。非关键性后端仍然可能会对请求截止时间过长的前端造成影响。前端不应该因此而拒绝大量请求、资源过限，或者延迟度大幅升高。

解决连锁故障的立即步骤

一旦检测到服务处于连锁故障的情况下，可以使用一些不同的策略来应对—同时，连锁故障是使用故障管理流程的好时机（更多信息参见第14章）。

增加资源

如果系统容量不足，而有足够的空闲资源，增加任务数量可能是最快的解决方案。然而，如果服务已经进入了某种死亡螺旋，只增加资源可能不能完全解决问题。

停止健康检查导致的任务死亡

某些集群任务管理系统，如Borg，周期性检查任务的健康程度，自动重启不健康的任务。可能健康检查自身反而成为导致任务失败的一种模式。例如，如果半数以上的任务由于正在初始化还不能够开始工作，而另外一半任务正在由于过载而无法服务健康检查。暂时禁止健康检查可能可以使系统恢复稳定状态。

进程任务的健康检查（“这个进程是否响应请求”）和服务级别的健康检查（“该进程是否能够回复这种类型的请求”）是两种概念不同的操作。进程任务的健康检查对集群管理系统有用，而服务健康检查对负载均衡器有用。清晰地区分这两种健康检查模式可以帮助避免这种场景。

重启软件服务器

如果软件服务器由于某种问题卡住了，而无法继续推进，重新启动可能会有帮助。针对以下场景可以试图重启：

- Java服务器处于GC死亡螺旋中。
- 某些正在处理的请求因为没有截止时间设置而正在消耗资源（如正在占用线程）。
- 死锁。

确保在重启服务之前先确定连锁故障的源头。还要确保这种操作不会简单地将流量迁移到别处。最好能够试验性地进行这种改变，同时缓慢实施。如果根本原因是因为冷缓存，那这种动作可能使现在的连锁故障更严重，

丢弃流量

丢弃流量是一个重型操作，通常是在连锁故障严重而无法用其他方式解决时才会采用。例如，如果高负载导致大部分服务器刚一启动就崩溃，可以通过以下手段将服务恢复到正常水平：

1.解决最初的触发原因（如增加容量）。

2.将负载降低到一定水平，使得崩溃停止。考虑在这里激进一些，如果整个服务都在崩溃循环中，那么可以考虑降低流量到1%的水平。

3.允许大部分的软件服务器恢复健康。

4.逐渐提升负载水平。

这个策略可以在负载恢复到正常水平之前帮助缓存预热，逐渐建立连接等。

显然，这样的操作会造成用户可见的问题。取决于该服务的配置，还应该看一下是否有办法可以（或者应不应该）差异化地丢弃用户流量。如果可以使用某种手段丢弃不重要的流量（例如预获取操作的流量），一定要先采用这种手段。

最重要的是，这个策略只有在底层问题已经修复的情况下才能恢复服务。如果触发这个连锁故障的问题没有修复（如，全局容量的紧缺），那么连锁故障可能在流量级别恢复之后再次发生。因此，在使用这个策略之前，应该先考虑修复（或者掩盖住）根源原因或者是触发条件。例如，如果一个服务内存不足，而进入了死亡螺旋，增加内存或者任务数量应该是首先要做的。

进入降级模式

通过提供降级回复来减少工作量，或者丢弃不重要的流量。这个策略必须要在服务内部实现，而且必须要求了解哪些流量可以降级，并且有能力区分不同的请求。

消除批处理负载

某些服务有一些重要的，但是并非关键的流量负载，可考虑将这些负载来源关闭。例如，搜索索引的更新、数据复制、请求处理过程中的资源统计等，可考虑关闭这些来降低负载。

消除有害的流量

如果某些请求造成了高负载，或者是崩溃（如致死请求），可考虑将它们屏蔽掉，或者是通过其他手段消除。

莎士比亚搜索服务的连锁故障

某个关于莎士比亚作品的纪录片在日本上映了，同时特别指明了莎士比亚搜索服务是进行进一步研究的最佳工具。随着这次广播，亚洲数据中心的流量激增，超过了服务容量。服务容量的问题伴随着当时正在进行的大型更新而变得更严重了。

幸运的是，一些安全防护措施帮助缓解了可能的故障。生产环境准备评审（production readiness review）流程指出了一些问题，开发团队已经解决。例如，开发者为服务加入了优雅降级功能。当容量不够时，服务不再返回照片，或者不再返回解释故事发生位置的小地图。取决于RPC的目的，超时的RPC要么不再重试（例如，之前提到的图片），要么采用随机指数型延迟进行重试。即使有这些保护措施的存在，任务还是一个接一个地失败了，然后被Borg系统重启，这导致正常工作的任务数量持续减少。

由于这个原因，服务监控页面上的某些图表变成了红色，并且SRE收到了紧急警报。为了解决这个问题，SRE临时向亚洲数据中心增加了一些服务容量，调整了莎士比亚搜索任务的任务数量。通过这种操作，成功恢复了亚洲数据中心的莎士比亚搜索服务。

接下来，SRE书写了一篇事后总结，详细说明了触发问题的事件，哪些做得好，哪些可以做得更好，和一系列待办事项来避免这个情景重现。例如，在服务过载的情况下，GSLB负载均衡器可以将一些流量导入邻近的数据中心。同时，SRE团队启用了自动伸缩机制，于是任务的数量可以自动跟着流量增长，这样他们就不用再操心这类问题了。

小结

当一个系统过载时，某些东西总是要被牺牲掉。一旦一个服务越过了临界点，服务一些用户可见错误，或者低质量结果要比尝试继续服务所有请求要好。理解这些临界点所在，以及超过临界点系统的行为模式，是所有想避免连锁故障的运维人员所必需的。

如果不加小心，某些原本为了降低服务背景错误率或者优化稳定状态的改变反而会让服务更容易出现事故。在请求失败的时候重试、负载自动转移、自动杀掉不健康的服务器、增加缓存以提高性能或者降低延迟：这些手段原本都是为了优化正常情况下的服务性能，但是也可能会提高大规模的服务故障的几率。一定要小心评估这些改变，否则灾难就会接踵而至。

第23章 管理关键状态：利用分布式共识来提高可靠性

作者：**Laura Nolan**

编辑：**Tim Harvey**

一个服务进程可能会在运行中崩溃，或者由于其他原因需要被重启。硬盘可能会出故障。自然灾难可能会同时让一个区域内的几个数据中心同时下线。SRE需要针对这些灾难做好事先准备，预先制定应对策略，以保障在灾难来临时系统仍能正常运行。这些应对策略经常需要将该服务分布式运行。跨物理区域分布式运行系统相对来说是比较简单的，但是却带来维护系统一致状态视图的需求。解决这个问题常常是比

较复杂且难以实现的。

一组服务进程可能想要可靠地对以下问题产生共识：

- 哪个进程目前是该组进程的领头人（leader）？
- 本组中都包含哪些进程？
- 是否已经将某个消息成功地插入了某个分布式队列？
- 某个进程目前是否还持有租约（hold a lease）？
- 数据存储中的某个键对应的值是什么？

在构建可靠的、高可用的系统过程时，我们发现分布式共识系统适合用来维护某个强一致的系统状态。这个分布式共识系统主要解决了在不稳定的通信环境下一组进程之间对某项事情达成一致的问题。例如，分布式系统中的几个进程可能会想对某个关键配置文件的内容，某个分布式的锁持有状态，或者某个队列中的消息是否已经被处理达成一致。该系统是分布式计算中的一个基本元素，几乎是Google所有服务都依赖的一个底层系统。图23-1 描绘了一组进程是如何利用分布式共识系统对某种系统状态保持一致的。

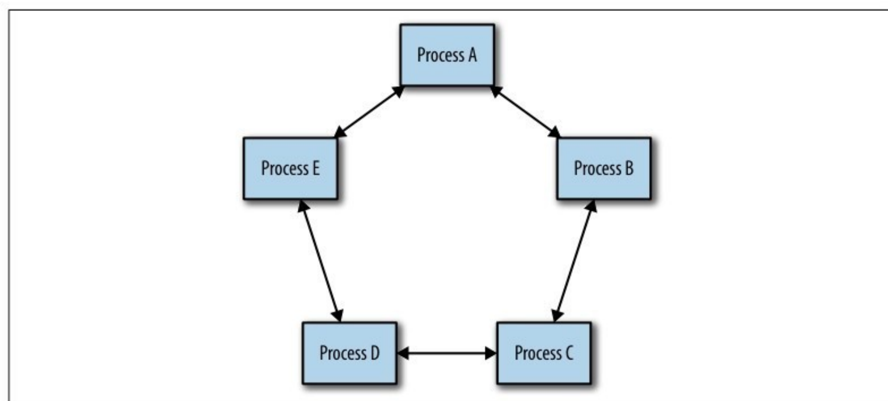


图23-1：分布式共识：一组进程达成一致

当你需要实现领头人选举（leader election）、关键性共享状态或分布式锁等时，我们建议采用某种正式证明过的、详尽测试过的分布式

共识系统来实现。如果不严肃对待这个问题很有可能会导致事故，在更糟的情况下，将造成某种非常难以修复的数据一致性问题，这可能会大大加长系统事故的处理时间。

CAP 理论

CAP理论（参见文献[Fox99]和[Bre12]）论述了一个分布式系统不可能同时满足以下三个要求：

- 每个节点上所见数据是一致的。
- 每个节点都可以访问数据。
- 可以承受网络分区问题（参见文献[Gil02]）。

该逻辑非常符合直觉：如果两个节点无法通信（因为网络出现了分区问题），那么整个系统要么在一个或多个节点上无法处理数据访问请求，要么可以照常处理请求，但是无法保障每个节点的数据具有一致性。

因为网络分区问题迟早会发生（光缆被切断，数据包由于拥塞造成丢失或延迟升高，硬件故障，网络组件配置错误等），理解如何构建分布式共识实际上就是理解某个服务应用如何实现强一致性和高可用。由于商业的压力，很多服务都需要很高的可用性，这些应用程序通常都需要对数据访问的强一致性。

系统工程师和软件工程师都很熟悉传统的ACID数据存储语义（原子性、一致性、隔离性和持久性），但是越来越多的分布式数据存储开始提供另外一套不同的语义，称之为BASE—基本可用、软状态、最终一致性（basically available、soft state、eventual consistency）。支持BASE的数据存储可以处理那些对支持ACID的数据存储来说成本特别高，甚至是压根不可能保存的超大数据集和事务。

大多数支持BASE语义的系统都依赖于多主复制（multimaster replication）机制，在这种模式下写操作可以并行在多个进程上执行，通过某种机制（常常是简单的“最后操作为准”）来解决冲突。这种方式

通常被称为最终一致。然而，最终一致可能会带来意想不到的问题（参见文献[Lu15]），尤其是当时钟漂移（在分布式系统中，这是不可避免的），或者网络分区（参见文献[Kin15]）发生的时候。[77]

同时，对开发者来说，针对仅仅支持BASE的数据存储来设计应用程序是很困难的。例如，Jeff Shute（参见文献[Shu13]）曾经说过，“我们发现开发者通常花费了大量的时间构建一个极为复杂和容易出错的机制来应对最终一致性下可能过时的数据。我们认为对开发者来说这是一种无法接受的负担，数据一致性的问题应该在数据库层解决。”

系统设计师不能通过牺牲正确性来满足可靠性或者性能的要求，尤其是在处理关键数据时。例如，假设某个系统处理财务交易：可靠性和性能在最终结果不正确的情况下一文不值。系统必须能够可靠地在多个进程中同步关键状态。分布式共识算法就提供了这种功能。

使用共识系统的动力：分布式系统协调失败

分布式系统通常很复杂，难以理解、监控和调试。运维这些系统的工程师通常会在灾难发生时对系统的行为所震惊。而灾难相对来说不那么容易发生，所以人们很少测试这些情景。在一次系统故障中，精确地解释该系统的行为是很困难的。尤其是在网络分区情况下—造成问题不一定是由完全分区导致的，而是：

- 网络非常慢。
- 某些消息可以通过，但是某些消息被丢弃了。
- 单方面的节流措施。

下一节提供了真实的分布式系统出现的问题的案例，同时讨论了如何使用领头人选举和分布式共识来预防这些问题的发生。

案例1：脑裂问题

某服务是一个文件存储服务，允许多个用户同时操作一个文件。该服务使用两组运行在不同机柜（rack）上的互相复制的文件服务器来提

高可靠性。该服务需要避免向某个复制组中的两台服务器同时写入数据，因为这样可能会造成数据损坏（可能是无法恢复的）。

每组文件服务器有一个领头者和一个跟随者，两组服务通过心跳互相监控。如果某个文件服务器无法联系到另外一个服务器，它会发送一个 STONITH（当头一枪）命令来强制关闭另外一个服务器，同时成为文件的主服务者。这种机制是业界减少脑裂（split-brain）场景发生的常规做法，但是接下来我们会论述，这在理论上是不正确的。

当网络变慢，或者开始丢包的时候会发生什么呢？在这个场景下，文件服务器会心跳超时，按照设计，它们会发送STONITH 命令给对方，同时成为文件的主服务者。但是，由于网络问题，某些命令可能没有成功发送。于是这两个文件服务器可能会存在同时为主的状态，也可能由于同时发送和接收到STONITH命令而都被关闭了。这要么会造成数据损坏，要么会导致数据不可用。

这里的根源问题在于，该系统正在尝试使用简单超时机制来实现领头人选举。领头人选举是分布式异步共识问题的另一种表现形式，它不能够通过简单心跳来正确实现。

案例2：需要人工干预的灾备切换

某个分片很多的分布式数据库系统的每个分片（shard）都有一个主实例，同步备份到另外一个数据中心的副实例上。某个外部系统检查主实例的健康度，如果主实例出现问题，将副实例提升为主实例。如果主实例无法确定副实例的健康度，那么它就会将自己标记为不可用，将问题升级给人工处理，以便避免案例1中的脑裂场景发生。

这种解决方案避免了数据丢失的危险，但是却影响了数据的可用性。同时，该系统没有必要地增加了运维人员的工作压力，运维压力实际限制了系统的扩展性。这类主从无法通信的问题在大型基础设施发生问题时很容易发生，运维人员可能已经忙着在解决其他问题了。如果网络质量真的很差，分布式共识系统也无法正确选举出主实例时，作为工程师可能也没有什么好办法来做出正确决策。

案例3：有问题的小组成员算法

某个系统有一个组件负责构建索引以及提供搜索服务。当启动时，各个节点使用谣言协议（gossip）来相互发现和加入某个集群。该集群选举出一个领头者，该领头者负责进行工作协调。当集群内部出现网络分区问题时，两个分区内部分别（错误地）各自选举出了一个领头者，同时接受写入和删除操作，从而造成了一种脑裂场景，造成了数据损失。维护一致的小组成员信息是分布式共识问题的另外一个变种。

事实上，很多分布式系统问题最后都归结为分布式共识问题的不同变种，包括领头人选举，小组成员信息，各种分布式锁和租约机制，可靠的分布式队列和消息传递，以及任何一种需要在多个进程中共同维护一致的关键状态的机制。所有这些问题都应该仅仅采用经过正式的正确性证明的分布式共识算法来解决，还要确保这个算法的实现经过了大量测试。任何一种临时解决这种问题的方法（例如心跳，以及谣言协议）在实践中都会遇到可靠性问题。

分布式共识是如何工作的

分布式共识问题有很多变种。当维护分布式软件系统时，我们关注的是异步式分布式共识（asynchronous distributed consensus）在消息传递可能无限延迟的环境下的实现。（同步式共识仅对实时系统有用，在实时系统里，有专门的硬件保障消息可以在特定时间内确保被送达。）

分布式共识算法可能是崩溃不可恢复（crash-fail）的（假设崩溃的节点再也不会返回系统中）或者崩溃可恢复（crash-recover）的。崩溃可恢复的算法更有用，因为大部分真实系统的问题都是临时性的，由于缓慢的网络或者是重启等原因造成的。

这种算法可能要应对拜占庭式（Byzantine）和非拜占庭式问题。拜占庭式问题指的是当某个进程由于程序Bug或者恶意行为发送不正确的消息的问题，这种问题相对来说处理成本更高，同时也更少见。

严格来讲，在有限时间内解决异步式分布式共识问题是不可能的。

正如Dijkstra 的获奖文章—FLP impossibility result（参见文献[Fis85]）写的那样，在不稳定的网络条件下，没有任何一种异步式分布式共识算法可以保证一定能够达成共识。

在实际操作中，我们通过保证给系统提供足够的健康的副本，以及良好的网络连接状态来保障分布式共识算法在大多数情况下是可以在有限时间内达成共识的。同时整个系统还需要加入随机指数型延迟。这样，我们可以保障重试不会造成连锁反应，以及本章后面会提到的角斗士（dueling proposers）问题。这个协议在一个有利环境下能够保障安全性，同时提供足够的冗余度。

最初的分布式共识问题的解决方案是Lamport的Paxos 协议（参见文献[Lam98]），但是也有其他的协议能够解决这个问题，包括Raft（参见文献[Ong14]）、Zab（参见文献[Jun11]），以及Mencius（参见文献[Mao08]）。Paxos本身也有很多变种尝试提升性能（参见文献[Zoo14]）。这些变种常常只是在某一个小细节上不同，例如给某个进程一个特殊的领头人角色以简化协议实现等。

Paxos概要：协议示例

Paxos 是基于提案（proposal）序列运行的，这些提案可能会被“大多数（majority）”进程所接受，也可能会被拒绝。如果某个提案没有被接受，那么它就是失败的。每个提案都对应有一个序列号，这就保证了系统的所有操作有严格的顺序性。

在协议的第一阶段，提案者（proposer）发送一个序列号给数个接收者（acceptor）。每个接收者仅仅在没有接受过带有更高序列号的提案情况下接受这个提案。提案者必要时可以用更高的序列号重试提案。提案者必须使用一个唯一的序列号（每个提案者从不相交的集合中提取序列号，或者将自身的主机名加入到序列号中等）。

如果提案者从“大多数”接收者那里接收到了“同意”回复，它便可以通过发送一个带有值的提交信息（commit message）来尝试提交这个提案。

提案的严格顺序性解决了系统中的消息排序问题。需要“大多数”参与者同意才能提交提案的要求保证了一个相同的提案无法提交两个不同的值，因为两个“大多数”肯定会至少重合一个节点。当接收者接收某个提案的时候，必须在持久存储上记录一个日志（journal），因为接收者必须在重启之后仍然保持这个状态。

Paxos本身来说不那么好用，它仅仅能够做到让节点共同接收一次某个值和提案号码。因为共同接收该值的节点数可以仅仅是“法定人数”（quorum）（也就是总数的一半再加1），任何一个节点可能都没有一个完整的视图，不知道目前已经被接收的所有的值。这个限制在大部分分布式共识算法中都存在。

分布式共识的系统架构模式

分布式共识算法是很底层、很原始的：它们仅仅可以让一组节点一次共同接受同一个值，这并不能很好地跟我们的设计任务相对应。真正使得分布式共识系统有用的是那些高级的系统组件，如数据存储、配置存储、队列、锁机制和领头人选举服务。这些基本的有实际意义的服务是分布式共识算法没有提供的。系统设计师需要这些高级组件，以降低系统设计的复杂度。同时这也使我们可以针对不同的环境或者要求更换不同的分布式共识算法。

很多成功使用分布式共识算法的系统常常是作为该算法实现的服务的一个客户端来使用的，例如ZooKeeper、Consul，以及 etcd。Zookeeper（参见文献[Hun10]）是第一个获得一定行业影响力的开源共识系统，因为它使用起来很方便，特别是对那些没有设计为使用分布式共识的系统来说。Google的Chubby 服务也很类似。Chubby的作者指出（参见文献[Bur06]），通过将共识系统作为一个服务原语提供，而非以类库方式让工程师链接进他们的应用程序，可以使得这些系统不需要为了高可靠共识服务而进行部署上的改变（例如运行正确数量的副本，处理小组成员问题，以及处理性能问题等）。

可靠的复制状态机

一个复制状态机（replicated state machine,RSM）是一个能在多个

进程中用同样顺序执行同样的一组操作的系统。RSM 是一个有用的分布式系统基础组件，可以用来构建数据和配置存储，执行锁机制和领头人选举（接下来会详细描述）。

在RSM系统上进行的操作是通过共识算法来全局排序的。这是一个非常有用的概念：多篇文章（参考文献[Agu10]、[Kir08]、[Sch90]）都提到了任何一个具有确定性的程序都可以采用RSM来实现成为一个高可用的、分布式复制的服务。

如图23-2所示，复制状态机是实现在共识算法逻辑层之上的一个系统。共识算法处理节点间对操作顺序的共识，RSM系统按照这个顺序来执行操作。因为小组中的每个成员在每次提案轮中不一定全部参与了，RSM可能需要在成员之间同步状态。在Kirsch和Amir（参见文献[Kir08]）的文章中描述到，可以采用滑动窗口协议（sliding-window protocol）在RSM成员之间同步状态信息。

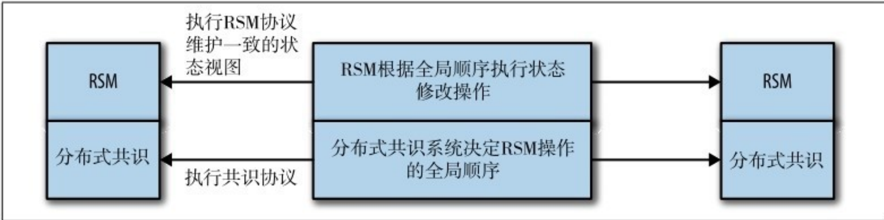


图23-2：共识算法和复制状态机的关系

可靠的复制数据存储和配置存储

可靠的复制数据存储是复制状态机的一个应用。复制数据存储在关键路径中使用到了共识算法。因此，性能、吞吐量和扩展能力在这种设计中非常重要。就像采用其他底层技术构建的数据存储那样，使用共识算法的数据存储可以对"读"操作提供多种不同的一致性语义，这在很大程度上决定了数据存储的扩展性。这些设计妥协将在本章后面的"分布式共识系统的性能问题"小节中进行详细讨论。

基于其他的非共识算法实现的系统经常简单地依赖于时间戳来决定返回哪些数据。时间戳在分布式系统中问题非常大，因为在多个物理机器上保证时间同步是不可能的。Spanner（参见文献[Cor12]）通过针对

最差情况下的不确定性建模，同时在必要时减慢处理速度来解决这个问题。

使用领头人选举机制实现高可用的处理系统

分布式系统的领头人选举是跟分布式共识等价的问题。复制多份服务并使用一个唯一的领头人（leader）来进行某种类型的工作是很常见的设计。唯一的领头人是一种保证粗粒度互斥性的方法。

这种设计类型在服务领头人的工作量是分片的，或者可以被一个进程所满足的情况下是合理的。系统设计师可以用一个简单的单机程序，复制几份，再采用领头人选举方式来保证任意时间只有一个领头人在运行（参见图23-3）的方式来构造一个高可用的服务。领头人的工作通常是负责协调某个工作者池中的工作者进程。这个模式被GFS（参见文献[Ghe03]）（已经被Colossus替代）以及Bigtable 键值对存储（参见文献[Cha06]）所采用。

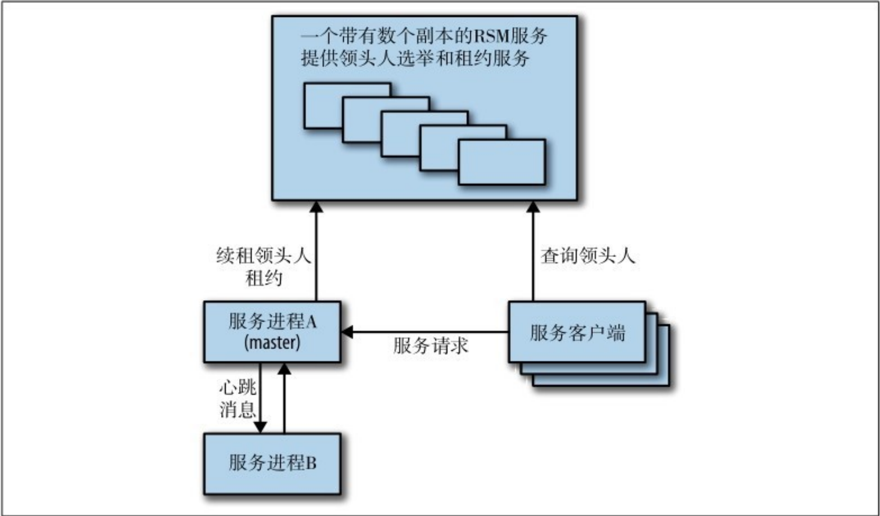


图23-3：使用复制进程进行领头人选举构造的高可用系统

在这种类型的组件里，不像复制数据存储那样，共识算法并不处于系统的关键路径中，所以共识算法的吞吐量不是系统的主要问题。

分布式协调和锁服务

屏障（barrier）在分布式计算中是一种原语，可以用来阻挡一组进程继续工作，直到某种条件被满足（例如，直到某个计算的第一阶段全部完成时，再继续进行）。使用屏障实际将一个分布式计算划分成数个逻辑阶段执行。例如，如图23-4所示，MapReduce（参见文献[Dea04]）可以使用屏障来确保整个Map阶段已经完成，再开始Reduce阶段的计算。

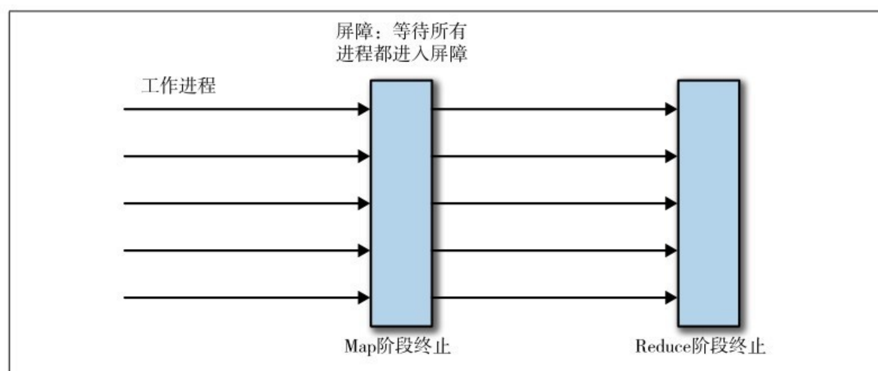


图23-4：在MapReduce计算中使用屏障协调进程

屏障可以由一个单独的协调者进程实现，但是这个实现会导致系统中出现不可接受的单点故障源。屏障也可以用一个RSM系统来实现。Zookeeper服务可以实现这种屏障模式：参见文献[Hun10]和[Zoo14]。

锁（lock）是另外一个很有用的协调性原语，可以用RSM实现。在一个分布式系统中，一些工作进程原子性地操作某些输入文件，同时将产生结果。分布式锁可以保障多个工作进程不会操作同一个输入文件。在实践中，使用可续租约（renewable Lease）而不是无限时间锁是很有必要的，避免某个进程崩溃而导致锁无限期被持有。分布式锁的具体实现超出了本章讨论的范围，但是要记住，分布式锁是一个应该被小心使用的底层系统原语。大多数应用程序应该使用一种更高层的系统来提供分布式事务服务。

可靠的分布式队列和消息传递

队列（queue）是一种常见的数据结构，经常用来给多个工作进程分发任务。

采用队列模式的系统可以很容易地处理某个工作节点的失效情况。然而，系统必须保证已经被领取的任务都被成功处理了。由于这个原因，建议采用某种租约系统（上述在讨论锁的时候已经讨论过），而不是单纯地从队列中取出任务。采用队列模式的一个问题是，如果队列不可用，整个系统都将无法工作。利用RSM来实现队列可以将危险性最小化，从而使得整个系统更加可靠。

原子性广播是分布式系统的一个原语，意思是整个系统的参与者都可以可靠地接收到消息，并且以同样的顺序来处理这些消息。这个概念在真实系统的设计中是非常有用的。市面上有大量的订阅-分发系统（publish-subscribe）可供系统设计师选用，但是并不是所有的系统都能够提供原子性的保障。Chandra 和 Toueg（参见文献[Cha96]）的文章证明了原子性广播和分布式共识本质上是一个问题。

这种使用队列进行任务分发的设计模式，其实是将队列作为一个负载均衡器来使用，正如图23-5所示，这种模式其实也可以被当作点对点通信来对待。通信系统通常也会实现某种订阅-分发队列，某条信息可能会被订阅这个信道或者主题的很多个客户端同时处理。在这种一对多案例中，队列中的消息被存储为一个持久化的有序列表。订阅-分发系统可以用来实现客户端订阅某种消息的机制，也可以用来实现一个分布式一致缓存。

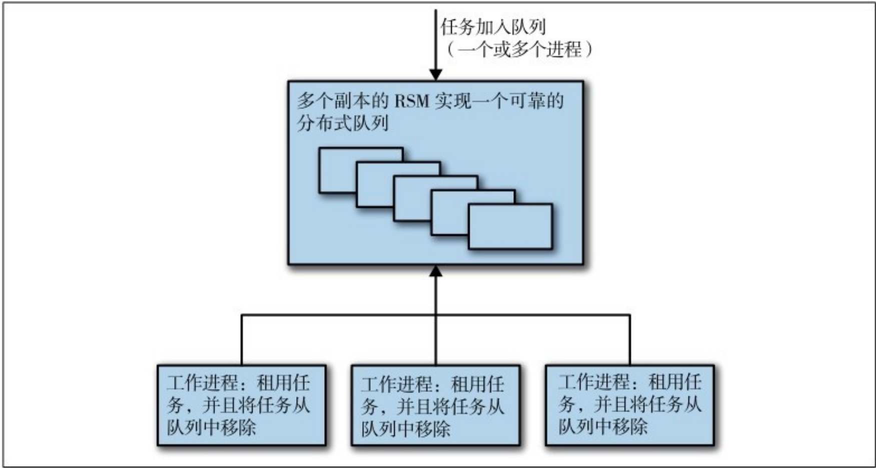


图23-5：一个基于队列的任务分发系统，利用分布式共识搭建的可靠队列。

队列和消息系统经常需要非常好的系统吞吐量，但是并不一定追求低延迟（这些系统很少是直接面向用户的）。然而，如果在上述这种多个工作进程从队列里获取任务的系统中系统延迟过高，将会导致系统处理能力大幅下降。

分布式共识系统的性能问题

传统理念经常认为分布式共识算法速度太慢，资源要求过高，不适合用于搭建高吞吐量并且低延迟的系统（参见文献[Bol11]）。这种观点是错误的—虽然某些分布式共识系统的实现比较慢，但是有很多办法可以提高性能。分布式共识算法是很多Google关键系统的核心，如文献[Ana13]、[Bur06]、[Cor12]和[Shu13]里提到的那样，这些系统在实践中证明分布式共识算法非常有效。要注意的是，Google的海量规模在这里可不是优势：我们涉及的数据集通常非常大，而且涉及的系统通常是跨地理区域部署的。大型数据集，再乘以几个副本意味着非常高的计算成本，同时跨地理区域部署的系统又会大幅增加副本之间的延迟，从而进一步降低系统性能。

世界上并没有某个性能“最优”的分布式共识和状态机复制算法，因为算法性能通常取决于与系统负载有关的多个因子，以及系统性能的目标和系统的部署情况。[78]下面几小节描述的大部分系统都是可用的并且在实际部署中的，有一些仍在科研阶段，这里提出是为了帮助读者更好地理解分布式共识系统的潜力所在。

系统负载（workload）可能会从多个维度大幅变动，理解负载变动的范围与特点是讨论系统性能的关键。在共识系统中，系统负载可能会从以下几个方面发生变动。

- 吞吐量：在负载峰值时，单位时间内提出提议的数量。
- 请求类型：需要修改状态的写请求的比例。
- 读请求的一致性要求。
- 如果数据大小可变，请求的大小。

部署策略也有很多可变之处，例如：

- 局部区域部署，还是广域部署？
- 采用的是哪种仲裁过程，大部分进程的分布情况如何？
- 该系统是否使用分片、流水线和批处理技术？

很多共识系统都会选举出一个指定的领头人进程，同时要求所有请求都必须发往该特殊节点。如图23-6所示，对处于不同地理位置的客户端来说，距离较远的节点有更高的往返周期（RTT），性能差距会非常大。

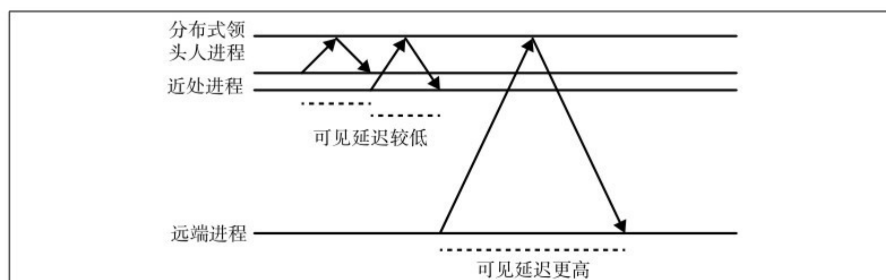


图23-6：对物理位置距离较远的客户端来说，距离带来的延迟效应很大。

复合式Paxos：消息流过程详解

复合式Paxos（Multi-Paxos）协议采用了一个强势领头人（strong leader）进程的概念：除非目前没有选举出任何的领头人进程，或者发生了某种错误，在正常情况下，提议者只需要对满足法定人数的进程发送一次消息就可以保障系统达成共识。这种强势领头人在很多共识协议中都适用，在系统需要传递大量消息的时候非常合适。

图23-7　展示了某个新的提议者（Proposer）进行协议的第一阶段Prepare/Promise部分的时候，系统的状态图。第一阶段成功后可以建立起一个新的有序视图（Numbered View），也就是一个领头人租约。在分布式共识协议后续阶段的执行中，该视图只要保持不变，就可以跳过协议的第一阶段，因为拥有视图的这个提议者可以简单地发送Accept消息，一旦收到多数参与者的回应（包括提议者自己）后，就可以保证系统达成了共识。

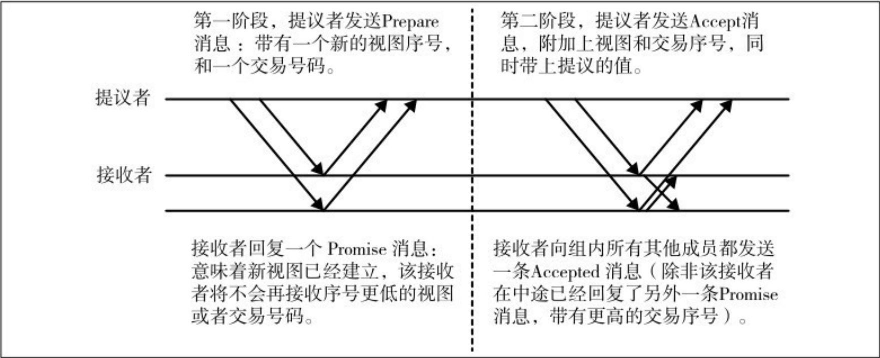


图23-7：基本的复合式Paxos 消息流

组内的另外一个进程可以在任何时间提交消息，并成为一个新的提议者，但是提议者角色的更换会带来性能上的损失。首先系统需要额外的时间重新执行协议的第一阶段部分，更重要的是，更换提议者可能会造成一种“提议者决斗”的状况。在这种状况下，两个提议者不停地打断对方，使得没有一个新的提议可以被成功接收。如图23-8所示，这种场景是一种活跃死锁的场景，可能会无限进行下去。

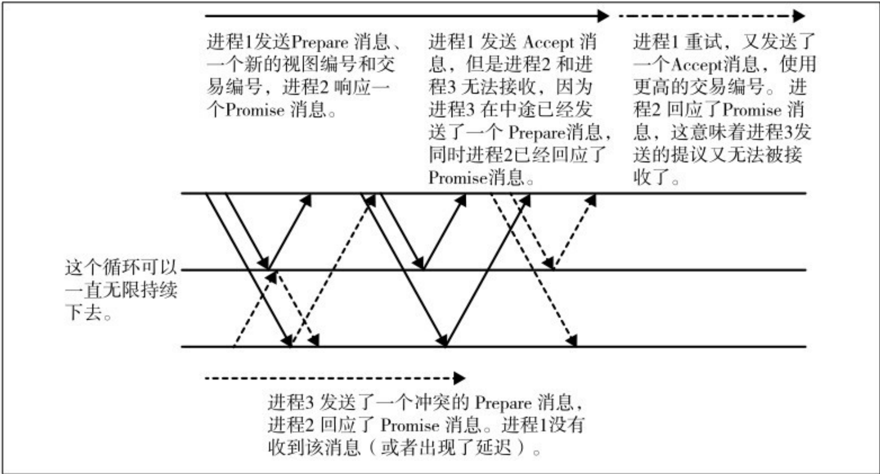


图23-8：复合Paxos协议中的“提议者决斗”场景

所有实用的共识系统都必须解决这种冲突问题，通常是要么选举一个固定的提议者进程（也就是领头人），负责发送系统中的所有提议，要么使用一个轮换机制，给每个进程划分特定的提议槽。

对使用领头人机制的系统来说，领头人选举机制必须仔细调优，以

在没有领头人的时候系统不可用，与多个领头人互相冲突的危险中取舍。设置正确的超时和后退策略是非常重要的。如果多个进程同时检测到目前没有领头人进程，同时试图成为领头人，那么很有可能没有一个进程能够成功（由于提议者决斗的问题）。在系统中引入随机变量是最佳选择。例如Raft（参见文献[Ong14]），就有一个非常完善的、考虑周全的领头人选举机制。

应对大量的读操作

许多系统都是读操作居多的，针对大量读操作的优化是这些系统性能优化的关键。复制数据存储的优势在于数据同时在多个地点可用，也就是说，如果不是所有的读请求都需要强一致性，数据就可以从任意一个副本来读取。这种设计对某些应用程序来说非常好，比如Google的Photon系统（参见文献[Ana13]）。该系统使用一个原子性的“比较”然后“设置”操作进行状态修改（受原子性寄存器的启发）。该操作必须要绝对一致，但是相比而言读操作就可以从任意一个副本进行，因为过期数据仅仅会造成额外工作，而不是错误的结果（参见文献[Gup15]）。这种取舍是非常值得的。

为了保障读取的数据是最新的，包含在该读取操作执行之前的所有改变，以下几条中的一条必须要满足：

- 进行一次只读的共识操作。
- 从一个保证有最新数据的副本读取数据。在使用稳定领头人进程的系统中（大部分分布式共识系统都会有），该领头人进程就可以提供这种保障。
- 使用法定租约（quorum lease）协议，在该协议下，某些副本被授予部分或者全部数据的一个租约，用一些写性能上的损失换来了强一致性的本地读操作的可能。该技术在接下来的一节中会详细描述。

法定租约

法定租约（qurom leases）（参见文献[Mor14]）是一个最新研究的分布式共识性能优化手段，该手段专注于降低操作延迟和提高读操作

的吞吐量。正如前文所述，在经典的Paxos协议和其他的分布式协议中，进行强一致性读操作（也就是该读操作的结果保证返回的是最新的系统状态）需要一次分布式共识操作，或者需要系统提供一个稳定的领头人进程。在很多系统中，读操作相比写操作要多得多，所以上述两种情况都极大地限制了系统的延迟和吞吐量。

法定租约技术针对数据的一部分给系统中的法定人数进程发放了一个租约，这个租约是带有具体时间范围的（通常很短）。在这个法定租约有限期间，任何对该部分数据的操作都必须要被法定租约中的所有进程响应。如果租约中的任何一个副本不可用，那么该部分数据在租约过期前将无法被修改。

法定租约对大量读操作的系统是非常有用的，尤其是当数据的某一部分是被集中在某一个地理区域的进程所读取的时候。

分布式共识系统的性能与网络延迟

分布式系统的写性能面临两个主要的物理限制。第一个是网络往返时间（RTT），另外一个是数据写入持久化存储的时间。我们接下来会先分析前者。

网络往返时间对不同的源地址和目的地址来说区别非常大，不仅由源地址和目的地址的物理距离决定，还包括网络拥塞程度。在一个数据中心中，两台机器之间的网络往返时间应该在毫秒级。在美国本土，两台机器的往返时间一般来说是45ms，而从纽约到伦敦则是70ms。

共识系统在局域网络中的性能和一个异步的领头人—追随者复制系统类似（参见文献[Bol11]），很多传统数据库都使用该系统进行复制操作。然而，分布式共识系统通常需要副本运行在“较远”距离上，这样可保障副本处于多个不同的故障域中。

很多共识系统使用TCP/IP作为通信协议。TCP/IP是面向连接的，同时针对消息的先进先出（FIFO）顺序提供了强保障。但是在发送任何数据之前，都需要先建立新的TCP/IP连接，也就是一次网络往返需要进行三次握手协议。TCP/IP慢启动机制也限制了连接的初始带宽。常见的

TCP/IP窗口大小在4～15KB之间。

TCP/IP的慢启动问题对共识组中的进程来说可能不是问题：因为这些进程会互相建立一个连接而且保持这些连接，因为通信将会很频繁。但是，对拥有非常多的客户端的系统来说，可能不能做到让每个客户端都和共识组所有进程都保持一个活动连接。因为TCP/IP需要消耗一定资源，包括文件描述符，以及对应的KeepAlive数据流量。对高度分片、拥有几千个副本的数据存储系统，以及更大规模的客户端来说，这种成本是无法接受的。一个解决方案是使用地域性的代理池，如图23-9所示。该代理池的进程与共识组建立持久的TCP/IP进程，以降低客户端的开销。同时代理池也是包装分片逻辑和负载均衡逻辑，以及共识系统的服务发现逻辑的好地方。

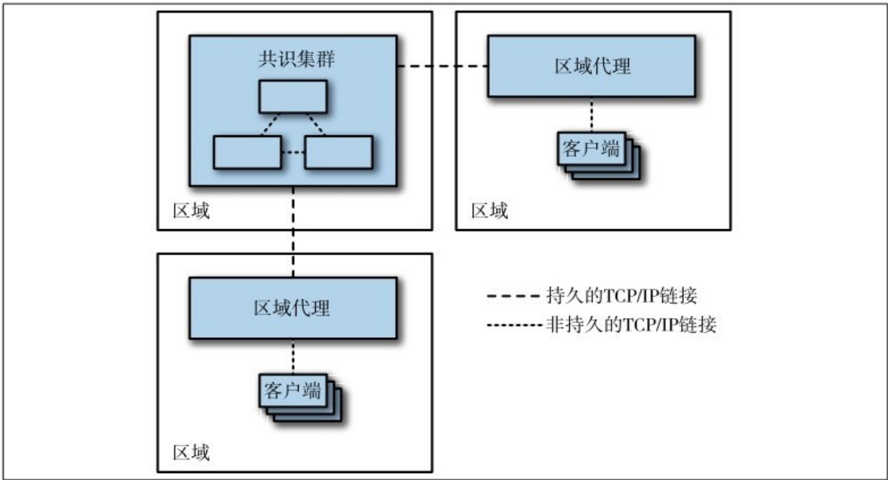


图23-9：使用代理池来降低客户端跨地理区域打开TCP/IP连接的开销

快速Paxos协议：性能优化

快速Paxos协议（Fast　Paxos，参见文献[Lam06]）是Paxos协议的一个变种，意在优化Paxos算法在广域网络中的性能。使用该协议的每个客户端可以直接向组内的接收者们发送Propose消息，而不需要像经典Paxos和复合Paxos那样通过一个领头人进程发送。这里的思路是用一个从客户端到所有接收者的并发消息来替代经典Paxos协议中的两个消息发送操作：

- 从客户端到提议者的一个消息。
- 从提议者到组内其他成员的一个并发消息。

直觉上来看，快速Paxos协议应该永远比经典Paxos协议要快。然而现实并不一定总是这样：如果Fast Paxos系统的客户端对接收者的RTT很高，而接收者之间却互相连接很快，那么我们其实是用N个慢速网络上的并发操作（快速Paxos协议）来替换一个慢速网络上的消息加上N个快速网络上的并发操作（经典Paxos）。由于延迟的长尾效应，在大部分的时间里，慢速网络上的一个操作远比一个分布式共识操作过程的速度要快（参见文献[Jun07]），因此快速Paxos协议在这种情况下反而比经典Paxos协议要慢。

许多系统将多个操作批处理化为一个操作发给接收者，以便增加系统吞吐量。让客户端直接作为提议者会使得批处理操作更困难。这是因为在这种情况下，各种提议将会独立地到达接收者处，使得批处理它们非常困难。

稳定的领头人机制

上文已经描述过，复合Paxos是如何通过选举一个稳定的领头人进程来提高性能的。Zab（参见文献[Jun11]）和Raft（参见文献[Ong14]）协议是其他两个例子，它们也通过选举稳定的领头人进程来提高性能。这种方案可以进一步进行读操作优化，因为领头人始终拥有最新信息，但是也存在以下几个问题：

- 所有的改变状态的操作都必须经由该领头人，这个要求使得距离领头人进程较远的客户端必须要增加额外的网络延迟。
- 领头人进程的外发网络带宽是系统性能的瓶颈（参见文献[Mao08]），因为该领头人的Accept消息包含了每个提议的所有数据，而其他进程发送的消息仅仅包含了交易数字，而没有真正的数据。
- 如果领头人进程恰好处于一台有性能问题的机器上，整个系统的吞吐量都会受到影响。

几乎在所有关注性能的分布式共识系统设计中，都采用了单个稳定领头人进程机制，或者是某种领头人轮换机制—预先分配给每个副本一个独立的分布式算法编号（通常是对交易编号取模）。使用轮换机制的算法包括Mencius（参见文献[Mao08]）和Egalitarian Paxos（参见文献[Mor12a0]）。

在广域网络中，在客户端分布在不同的地理区域而共识组分布相对集中的情况下，这种领头人选举机制可以使客户端所见的延迟降低，因为领头人与共识组成员的RTT将会比客户端到各成员的RTT要低。

批处理

批处理机制，如上一页“快速Paxos协议：性能优化”一节所描述的那样，可以增加系统吞吐量。但是即使这样，某个共识组副本在等待其他人回复的时候仍然是处于闲置状态的。针对这种情况，我们可以采用流水线机制，从而使得多个提议可以同时进行。这种优化与TCP/IP的滑动窗口机制很像。流水线机制通常会和批处理机制结合使用。

流水线中的一批请求仍然是用一个视图序号和交易序号全局排序的，所以这个方法并不会违反复制状态机所需要的全局排序机制。针对这种优化的详细讨论请参见文献[Bol11]和[San11]。

磁盘访问

为了使某个崩溃的节点返回集群后仍能够记住崩溃之前的状态，该节点需要将请求记录到持久化存储中。例如，在Paxos协议中，某个接收者如果已经接收过一个更高序列号的提议，就不能再接收低序列号的提议了。如果之前接收的和承诺过的记录没有被记录到持久化存储中，那么该接收者就可能会违反协议要求，从而造成数据不一致的情况。

将某条记录写入磁盘上的记录的时间根据所用硬件或者所用的虚拟化环境差距是很大的，但是总体来说，大致在1毫秒到数毫秒之间。

复合Paxos的信息流的过程在本章前面“复合式Paxos：消息流过程详解”一节中描述过了，但是并没有展示在什么情况下协议要求必须在磁盘中记录状态改变。在进程做任何一个承诺之前都必须进行磁盘写操

作。在复合Paxos性能的关键点—协议的第二部分中，这些操作点处于接收者针对某个提议发送Accepted消息之前，以及提议者发送Accept消息之前—因为这条Accept消息其实是一个隐含的Accepted消息（参见文献[Lam98]）。

这就意味着在单个共识操作的延迟中，有以下几个操作：

- 提议者的一次硬盘写入操作。
- 并行消息发送给接收者。
- 每个接收者的磁盘写操作（并行）。
- 回复消息的传递。

有一个版本的复合Paxos协议在磁盘写操作中占主要成分的时候适用：该变体协议并不认为提议者的Accept消息是一个隐含的Accepted消息。在这个协议下，提议者和其他进程同时并行写入磁盘，并且单独再发送一次Accept消息。于是，在这种情况下，系统延迟就和发送两条消息，以及法定人数进程并行进行磁盘写操作成比例了。

如果对磁盘的一次小型随机写操作在10ms数量级上，那么共识操作将会被限制为大概每分钟100次。这些数字假设网络往返时间是可以忽略的，并且提议者是与其他接收者并行进行写操作的。

正如上文所述，分布式共识算法经常用来实现一个复制状态机。RSM需要保留一份交易日志，以便用于灾难恢复（正如其他数据存储那样）。共识算法的日志可以和RSM的交易日志合为一个。将这两个日志合并可以避免不停地向磁盘上的两个不同位置交替写入（参见文献[Bol11]），也就降低了磁盘寻址操作消耗的时间。这些磁盘于是可以每秒处理更多操作，该系统也可以每秒处理更多操作。

在数据存储系统中，磁盘在维护日志之外还有其他用处：数据通常也是存放于磁盘中的。日志的写操作必须直接刷写到硬盘，但是数据的改变通常可以写入内存缓存中，稍后再写入磁盘，可以进行重新排序以便更有效地写入（参见文献[Bol11]）。

另外一个可能的优化点是在提议者中将多个客户端的操作批处理为一个操作（参见文献[Ana13]、[Bol11]、[Cha07]、[Jun11]、[Mao08]和[Mor12a]）。这将磁盘日志的写开销与网络延迟分布到更多操作上，从而提高系统吞吐量。

分布式共识系统的部署

系统设计者部署共识系统时，最重要的决策在于选择副本的数量和对应的部署位置。

副本的数量

一般来说，共识系统都要采用“大多数”的法定过程。也就是说，一组2f+1副本组成的共识组可以同时承受f个副本失败而继续运行（如果需要容忍拜占庭式失败，也就是要能够承受副本返回错误结果的情况，则需要3f+1个副本来承受f个副本失败（参看文献[Cas99]））。针对非拜占庭式失败的情况，最小的副本数量是3—如果仅仅部署两个副本，则不能承受任何一个副本失败）。3个副本可以承受1个副本失败。大部分系统的不可用时间都是由计划内维护造成的（参见文献[Ken12]）:3个副本使该系统可以在1个副本维护时继续工作（这里假设其余两个副本可以承受系统负载）。

如果某个非计划内的失败情况在常规维护时发生了，那么共识系统就会不可用。而共识系统的不可用通常是不可接受的，于是在这种情况下，我们需要5个副本同时运行，于是可以允许系统在两个副本不可用的情况下继续运行。如果5个副本中有4个能正常运行，则不需要进行任何人工干预操作，但是如果仅仅只有3个能正常运行，那么我们需要立刻增加一个或者两个额外的副本。

如果共识系统中大部分的副本已经无法访问，以至于无法完成一次法定进程，那么该系统理论上来说已经进入了一个无法恢复的状态。因为至少有一个副本的持久化日志无法访问。在这个状态下，有可能某个操作仅仅只在无法访问的副本上执行了。这时，系统管理员可以强行改变小组成员列表，将新的副本添加到小组中，使用其他可用成员的信息来填充。但是数据丢失的可能性永远存在—这种情况应该全力避免。

在灾难处理过程中，系统管理员需要决定是否需要进行这种强制性的重新配置，或者仅仅是等待那些机器恢复可用。当决策做出后，对系统日志的处理（以及对应的监控）就变得非常重要。理论性的论文常常指出共识系统可以用来构建一个分布式日志，但是却不会讨论如何应对某个崩溃后又恢复的副本（这些副本可能缺少某一段共识信息）或者是某个速度慢的副本。为了保证系统的鲁棒性，副本必须要进行某种消息同步，以便获得最新信息。

分布式日志并不总是分布式共识系统理论的一部分，但却是生产系统中很重要的一部分。Raft描述了一个管理分布式日志的方法（参见文献[Ong14]），准确地定义了在分布式日志中的各种空洞应该如何填补。如果5个实例的Raft系统丢失了除领头人进程之外的其他进程，领头人进程仍然能保证拥有全部已经提交的提议的信息。反之，如果丢失的大多数实例中包含领头人实例，那么就无法保障其他副本的信息及时性。

系统性能和仲裁过程中不需要的副本数量有直接关系：系统中的一小部分副本可以落后，从而使得其他参与仲裁过程的副本跑得更快（只要领头人进程性能良好即可）。如果副本之间的性能差距很大，那么系统副本的任何一个失败都会影响性能，因为这时慢的副本也必须要参与仲裁过程。系统能够承受的失败和落后副本的数量越多，那么整体的系统性能就可能越好。

成本也是一个管理副本时需考虑的因素：每个副本都需要很多的计算资源。如果所讨论的系统是一个简单集群，那么运行副本的资源可能就不是一个很大的考虑点。然而，如果副本的运行成本很高，例如Photon（参见文献[Ana13]），该系统使用分片配置，其中每个分片都使用共识算法组成一组。随着分片数量的增多，每个额外副本的数量也会增多。

于是，针对任何系统的副本数量的考虑都是基于以下几个因素中的一个进行妥协：

- 对可靠性的要求

- 计划内维护操作的频率
- 危险性
- 性能
- 成本

最后的决策每个系统都会不同：每个系统都有不同的可用性服务水平目标；某些组织会更频繁地进行维护性操作；而某些组织使用的硬件成本、质量和可靠性都有所不同。

副本的位置

关于共识集群进程的部署位置的决策主要来源于以下两个因素的取舍：1.系统应该承受的故障域数量。2.系统的延迟性要求。在选择副本位置时，会产生很多复杂问题。

一个故障域（failure domain）是指系统中可能由于一个故障而同时不可用的一组组件。常见的故障域包括：

- 物理机器。
- 数据中心中用同一个供电设施的一个机柜。
- 数据中心中的数个机柜，使用同一个网络设备连接。
- 受单个光纤影响的一个数据中心。
- 处于同一个地理区域的一组数据中心，可能被同一个自然灾害所影响。

一般来说，随着副本之间的距离增加，副本之间的网络往返时间也会增加，但是系统能够承受的失败程度也会增加。对多数共识系统来说，网络往返时间的增加会导致操作延迟的增加。

对延迟的敏感性和对灾难的承受程度，每个系统很不一样。在某些共识系统架构下并不需要特别高的吞吐量，或者特别低的延迟：例如，

某个共识系统如果只是为了提供成员信息和领头人选举，服务一般不会负载很高，如果共识操作时间仅仅是领头人租约的一小部分时，性能并不是关键点。批处理居多的系统也很少会被延迟所影响：可以通过批处理数量的增加来提高吞吐量。

有的时候，不断增加系统可承受的故障域大小并不一定合理。例如，如果共识系统的所有客户端都在某个故障域内（例如，纽约州范围内），虽然在广域范围内部署一个分布式系统可以使得在这个故障域出现故障时继续工作（例如，龙卷风Sandy发生时），但是这是不是值得呢？很有可能不值得，因为该系统的所有客户端都无法工作了，该系统没有实际价值。这种额外的成本，不管是延迟上的、吞吐量上的，还是计算资源上的付出都没有任何回报。

我们考虑副本位置时，应该将灾难恢复考虑在内：在某个存储关键数据的系统中，共识系统的副本也就是该系统数据的在线拷贝。然而，当处理关键数据时，即使已经有了可靠的共识系统部署在不同的故障域范围内，我们也必须经常将数据备份在其他地方。因为有两个故障域是我们永远无法逃避的：软件本身和系统管理员的人为错误。软件系统中的Bug可能会在罕见情况下浮现造成数据丢失，而错误的系统配置也可能会造成类似情况。而人工操作可能会执行错误的命令，从而造成数据损坏。

当决定副本位置的时候，记得最关键的因素是客户端可见性能：理想情况下，客户端和共识系统之间的RTT应该是最小化的。在广域网络中，无领头人的协议，例如 Mencius和 Egalitarian Paxos可能有一定的性能优势，尤其是当应用程序设计为可以对任意副本进行读操作而不需要进行共识操作时。

容量规划和负载均衡

当设计某个部署场景时，我们必须保证有足够容量应对系统负载。在分片式部署时，可以通过调整分片的数量来调整容量。然而，对那些可以从副本直接读取数据的系统来说，可以通过添加副本来提高读性能。增加更多的副本也有成本：在使用强势领头人过程的算法中，增加副本会增加领头人进程的负载，在P2P协议中增加一个副本则会增加其

他所有进程的负载。然而，如果写操作有足够容量，而大量的读操作正在给系统造成压力，增加副本可能是最好的选择。

这里需要说明的是，向一个采取“大多数”法定仲裁过程系统中增加新的副本可能会降低系统的可用性，如图23-10所示。对ZooKeeper和Chubby来说，一个常见的部署使用5个副本，一个法定仲裁过程需要3个副本参与。整个系统可以在两个副本，也就是40% 不可用的情况下仍然正常工作。当使用6个副本时，仲裁过程需要4个副本：也就是超过33% 的副本不可用就会导致系统无法工作。

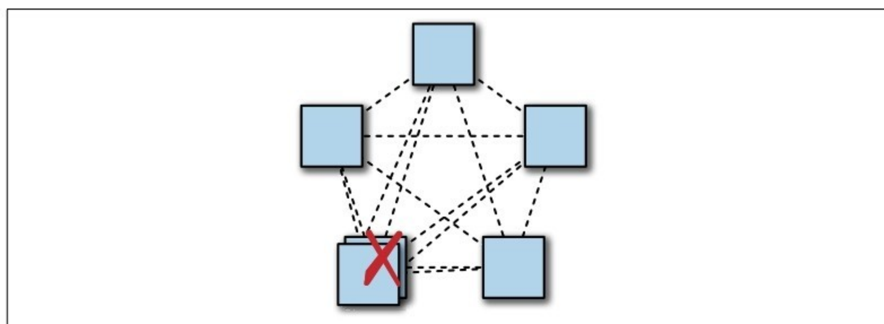


图23-10：向系统中一个区域增加一个额外的副本可能会影响系统可用性。

将多个副本放在同一个数据中心中可能会减少系统可用性：这里展示了当该数据中心出现故障时整个共识组没有任何备用副本可用。

在添加了第6个副本时，针对故障域的考虑就更重要了：如果某个组织有5个数据中心，并且通常在5个数据中心中各运行1个共识组副本，那么某个数据中心的故障仍可以在其他组中留下某个备用副本。如果第6个副本也被部署到这5个数据中心中的一个，那么该数据中心的故障会同时造成该共识组的两个备用副本同时不可用，也就是将容量减少了33%。

如果客户端集中于某个物理区域，那么最好将系统副本放置在离客户端近的地方。然而，当考虑究竟将副本放在哪里时，我们还需要考虑负载均衡机制，以及系统如何应对过载情况。如图23-11　所示，如果一个系统简单地将读请求发往最近的副本，那么在某个区域的一次负载升高就会导致最近的副本过载，接着将第二近的副本吞没，以此类推，最终造成连锁故障。这种过载通常是由批处理任务触发的，尤其是当多个

批处理任务同时启动时。

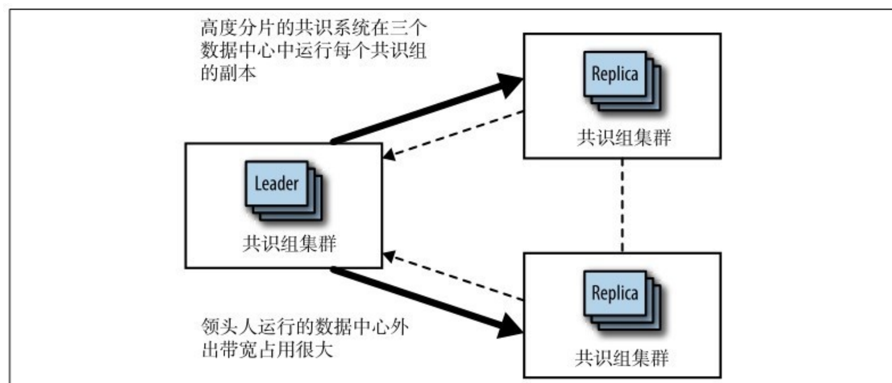


图23-11：领头人进程放置在一起会导致不均衡的带宽使用

我们已经讨论过很多分布式共识系统采用领头人机制来提高性能。然而这里要注意，领头人进程会使用更多的计算资源，尤其是网络容量。这是因为，领头人进程会发送带有数据的提议，但是其他副本发送的都只是回应数据，相对较小。运行高度分片的共识系统的组织可能会发现，保证领头人进程在数据中心之间分布相对均衡是很有必要的。这样做可以保证系统不会由于一个数据中心的出口造成瓶颈，于是可以提升系统的整体性能。

将共识组置于不同的数据中心（参见图23-11）中的另外一个劣势是当领头人所处的数据中心出现大规模故障时，整个系统会剧烈变动（如电源故障、网络设备故障、光纤被切等）。如图23-12　所示，在这个灾难场景中，所有的领头人都应该切换到另外一个数据中心，要么是平均分配，要么会涌入同一个数据中心。在这两种情况中，其他两个数据中心的网络流量会突然增大。这时如果发现数据中心之间的带宽不够，那么就太不幸了。

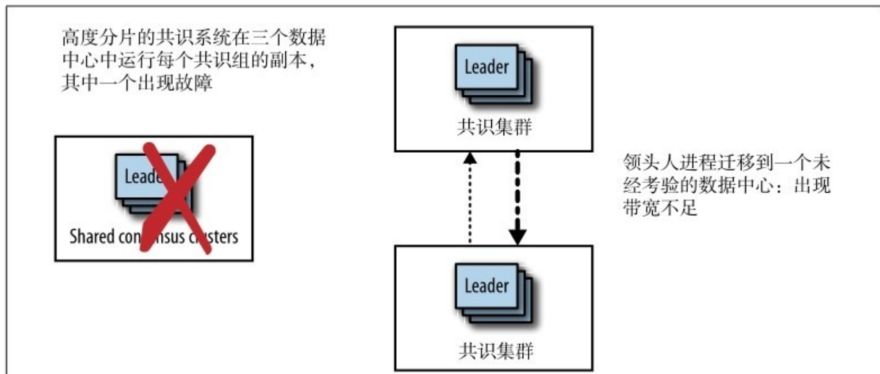


图23-12：当放置在一起的领头人进程同时迁移时，网络流量的使用率会大幅变化。

然而，这种场景也可能由于自动化领头人选举机制造成，例如：

- 客户端通常在领头人距离更近时，有更好的延迟。那么选择领头人位置的算法可能会采用这个信息。

- 某个算法可能会将领头人置于性能最好的机器上。这里的问题在于，如果三个数据中心中的某个数据中心有更好的机器，那么当这个数据中心出现问题时，将会有剧烈的网络流量变动。为了避免这个问题，该算法必须同时考虑到分布的均衡性，以及机器的能力。

- 领头人选举算法可能会优先运行时间长的进程。如果软件更新是按数据中心来发布的，那么长时间运行的进程很可能会聚集在一起。

仲裁组的组成

当决定在哪里放置共识组的副本时，另外一个重要因素是要考虑地理位置的分布（或者更准确地说，副本直接的网络延迟）对性能的影响。

一个做法是将副本分布得最均匀化，使得副本之间的RTT基本类似。在其他因素都相同的情况下（如负载、硬件、网络性能等），这种分布应该会在各个地区都形成相对固定的系统性能，不管该组的领头人被放在了何处（在无领头人的协议中，各个成员性能也应该类似）。

地理位置的分布可能会给这种做法造成一定难处，尤其是当跨大陆

与跨大西洋和跨太平洋的流量对比的时候。如果考虑跨越北美洲和欧洲部署的某个系统：试图平均分布副本是不可能的，系统中永远会有一段延迟高的链路，因为跨大西洋的链路要比跨大洲的链路速度慢。无论如何，在某个区域中的操作都要跨越大西洋进行一次共识操作。

然而，如图23-13所示，为了将系统流量分布得更为均匀，系统设计者可以考虑采用5个副本，将其中2个副本放置于美国中心地带，1个放置于东海岸，另外两个放置于欧洲。这样分布可基本保证共识过程在北美洲的副本上完成，而不需要等待欧洲的回复。而从欧洲起始的共识过程可以仅仅跟美国东海岸的副本完成。东海岸的副本就像两个可能的仲裁组的关键轴，将两个组连接在了一起。



图23-13：两个重叠的仲裁组使用一个副本作为连接

如图23-14　所示，该副本的故障可能会导致系统延迟大幅改变：以前系统主要是受美国中部到东部的RTT影响，或者欧洲到东海岸的RTT影响，现在则会成为受欧洲到美国中部的RTT影响，也就是比之前有50%的增加。在这种情况下，距离最近的仲裁组的物理距离和网络延迟都会受到很大影响。

这种情况是简单多数型仲裁过程在成员RTT非常不同的时候的一个关键弱点。在这种情况下，层级型的仲裁过程可能更有用。如图23-15所示，9个副本可能会被部署为3组，每组3个。仲裁过程可以由多数组完成，而每个组只有在多数成员可用的情况下才可用。这意味着一个副本可以在中央组中出现故障，而不会对系统整体性能产生影响，因为中

央组仍有两个可用副本，仍可以参与仲裁。



图 23-14：链接性副本的故障会立刻导致任何仲裁组的RTT大幅增加



图23-15：层级型仲裁过程可以用来减少对中央副本的依赖程度

当然，运行更多数量的副本需要更高的成本。在高度分片、大量读操作可以被副本服务填充的系统中，我们可以通过使用更少的共识组来应对这种成本。这样的策略意味着系统中总共的进程数量不会改变。

对分布式共识系统的监控

正如上文所述，分布式共识算法是很多Google关键系统的核心（参见文献[Ana13]、[Bur06]、[Cor12]和[Shu13]）。所有的重要生产系统为了更好地检测故障或者进行故障排除，都需要监控。经验告诉我们，分布式共识系统的某些区域需要特别关注，包括以下几项。

每个共识组中的成员数量，以及每个成员的状态（健康或不健康）

某个进程可能仍在运行，但是却由于某种原因（硬件因素）无法工作。

始终落后的副本

健康的共识组成员可能仍然处于不同的状态中。某个组成员可能是在启动时从同伴处恢复状态信息，或者相比仲裁组成员始终处于落后状态，或者目前处于完全参与状态，或者是当前组的领头人角色。

领头人角色是否存在

基于类似复合Paxos算法的系统使用领头人角色，该角色必须被监控，以便保障领头人存在，因为如果系统不存在领头人角色，系统则不可用。

领头人角色变化的次数

领头人角色的快速变化会影响那些使用稳定领头人角色的共识系统的性能，所以领头人角色变化的次数应该被监控。共识算法通常使用新的租约或者新的视图编号来标记领头人角色的变动，所以这可能是一个比较好的监控指标。领头人角色的快速改变意味着领头人正在快速变换（flapping），可能是由于网络连接问题导致的。而视图编号的降低，可能预示着软件中的一个严重Bug。

共识交易记录数字

系统管理员需要知道目前系统是否正在处理交易。大多数共识算法采用一个递增的共识交易数字来代表目前系统的进度。这个数字在系统健康时应该随着时间不断增长。

系统中的提议数量，以及系统中被接受的提议数量

这些数字可以显示当前系统是否在正确运行。

吞吐量和延迟

虽然这并不具体针对某个分布式共识系统，但这些特性都应该被系统管理员监控和理解。

为了更好地理解系统性能，以及帮助进行故障排查，我们还要监控以下几点：

- 针对提议接收时间的延迟分布。
- 系统不同部分之间观察到的网络延迟。
- 接收者在持久化日志上花费的时间。
- 系统每秒处理的字节数。

小结

本章我们探索了分布式共识问题的定义，同时展示了几种常见的分布式共识系统架构，以及这些系统对应的性能特点和部分运维重点。

本章故意避开了关于特定算法、协议和实现的深入讨论。分布式协调系统和对应的底层技术正在飞速发展，这些信息会很快过时，但是这里讨论的基础知识，以及所引用的文章则会使读者不管在使用今天可用的分布式协调工具时，以及未来的软件时都能用得上。

如果读者仅仅从本章中学到一件事，那么就应该记住分布式共识系统可以用来解决的问题类型，以及使用类似心跳机制而不是分布式共识机制所造成的问题。每当读者见到领头人选举、关键共享状态，或者分布式锁的时候，一定要想起分布式共识：任何其他的方案都是系统中的一枚定时炸弹，随时可能爆炸。

第24章 分布式周期性任务系统

作者：**Štěpán Davidovič**[79]

编辑：**Kavita Guliani**

本章描述了Google所实现的一个分布式周期性任务系统（Cron），该系统为Google内部绝大多数团队的周期性任务提供服务。看似一个简单的基础服务，但是随着该系统的演变我们从中学到了很多宝贵的经验和教训。在这一章里，我们会讨论到分布式周期性任务系统面临的一些问题，以及可能的解决方案。

Cron，作为一个很常见的UNIX工具，设计目标是根据用户定义的时间或者间隔来周期性启动一个任意的任务。下文我们会首先分析Cron的核心思想以及常见的实现方式，接下来会讨论这样一个应用程序如何能在一个大型、分布式环境下工作，以避免单个物理机器造成的单点故障。我们描述了一个分布式的Cron实现，该系统运行于很少的几台机器上，但是却可以利用一个数据中心任务分发系统（例如，Borg，参见文献[Ver15]）来在整个数据中心中运行大量任务。

Cron

我们首先来讨论Cron的常用场景—单物理机场景，然后再讨论跨数据中心的实现。

介绍

Cron系统在设计上允许系统管理员和普通系统用户指定在某时某刻运行某个指令。Cron可以执行很多种类型的任务，例如垃圾回收类任务和周期性的数据分析任务。最常见的执行时间表格式是所谓的“crontab”格式，这个格式支持使用简单的周期定义（例如，“每天中午执行一次”、“每小时整点的时候执行”）。也可以通过这种格式进行复杂的周期定义，例如，“仅仅在每月第30天，且当天为周六的时候执行”。

Cron　通常的实现方式是由一个单独组件组成，通常被称为crond。crond是一个系统守护进程（daemon），该进程负责加载Cron任务的定义列表，而任务会在对应的时间被启动。

可靠性

从可靠性的角度来看，Cron服务有几个方面值得一提：

- Cron 服务的故障域仅仅是单台物理机器。如果这台物理机器并没有在运行，那么Cron服务和对应的用户任务都不会运行。[80]这里我们如果使用一个非常简单的模型，将Cron服务运行在一台机器上，那么可用远程方式在另外一台物理机器上启动工作任务（例如，使用SSH）。这个场景中有两个不同的故障域，任何一个故障域的故障都可能影响整个服务的可靠性。

- 当crond重启时（包括物理机器重启），唯一需要保存的状态就是crontab配置文件自身。Cron进程启动任务之后不会再追踪任务状态。

- anacron 是一个特例。anacron 在系统恢复运行时，会试图运行那些在宕机时间中本来应该运行的程序。但是这种重新运行机制只会对那些每天运行一次或者更短周期的任务起作用。这种功能对运行在工作站和笔记本上的维护性任务来说非常有用。该功能是通过将所有已注册的Cron任务上次运行的时间记录在一个文件中来实现的。

Cron任务和幂等性

Cron任务一般都被用来执行周期性任务，但是除此之外我们很难知道某个任务的具体功能。数量繁多、功能和需求各异的Cron任务很明显会影响到整个服务的可靠性要求。

有些Cron任务，例如垃圾回收机制，是具有幂等性的。在系统异常状态下，多次运行该任务也是安全的。但是其他Cron任务，例如某个大批量发送E-mail通知的任务，是不应该运行多于一次的。

让整个情况变得更复杂的是，对于某些任务来说没有运行也是可以接受的，但是对其他的任务来说却不是这样。例如，某个垃圾回收任务定于每5分钟运行一次，可能可以偶尔跳过一次执行。但是一个每月执行一次的工资计算任务却不应该被跳过。

Cron任务的这种多样性使得讨论系统的可靠性变得很难：针对Cron这样的服务，并没有一个可以适用所有情况的固定答案。一般来说，在系统允许的情况下，我们更倾向于最差情况下跳过某个任务的执行，而不会冒有可能将任务执行两次的风险。这是因为通常情况下，试图修复

某个任务没有执行的问题，要比修复任务执行两次造成的问题容易。Cron任务的所有者，可以（并且应该）监控他们自己的Cron任务：例如，某个任务的所有者可以通过Cron服务暴露的状态信息监控自己的任务，也可以建立起独立的、针对Cron任务执行效果的监控。在任务没有执行的情况中，任务所有者可以针对该任务采取合适的行动。与之相比，之前提到的发邮件任务，如果试图从运行两次的情况中恢复经常是很难的，甚至是不可能的。因此，在Cron服务中，我们优先于“失效关闭（fail-close）”，以此来避免系统化地制造困难局面。

大规模Cron系统

从单台物理机器的部署场景迁移到更大规模部署的场景需要我们重新思考Cron任务的某些基础定义，这样才能使得Cron在大规模场景下工作得更好。在展示Google的解决方案之前，我们会先讨论小规模部署和大规模部署的区别，并且描述在大规模部署下必需的某些设计改变。

对基础设施的扩展

在常规实现里，Cron只能运行在一台机器上。大规模系统部署需要将Cron分布到多台机器上。

将Cron服务部署在单台物理机器上对可靠性来说可能是灾难性的。假设这台物理机位于一个拥有1000台物理机的数据中心中，那么一次1/1000的不可用故障就可能使得整个Cron服务不可用。这种实现显然是不可行的。

为了提升Cron的可靠性，我们将实际进程与物理机器分开。如果我们想要运行一个服务，仅仅通过指明该服务的资源要求，以及应该运行的数据中心即可。数据中心内部的任务分发系统（该系统应该是高可靠的）来决定一台或者多台机器来运行你的服务，同时该系统还会处理物理机的故障问题。这样一来，运行一个任务就仅仅等于向数据中心分发器发送一个RPC了。

然而，这个过程是需要一定时间的。一台损坏的物理机需要一定时间才能被发现，通常是在某项健康检查超时时被发现，然后任务分发系

统会将你的服务迁移到另外一台机器上，消耗一定时间重新安装软件并且启动新的进程。

因为进程迁移到另外一台虚拟机上的过程可能会导致存储在原有机器上的所有本地状态丢失（除非采用某种在线迁移技术）。同时整个系统的重新分发过程可能会超过系统所定义的最小时间间隔：1分钟。于是我们需要一些针对数据丢失和重新启动时间过长的应对措施。为了保持本地状态，我们可能会简单地将状态存储在一个分布式文件系统中，例如GFS。在服务启动的时候，直接通过该分布式文件系统查询并且启动那些在系统迁移过程中没有运行的任务。但是这种解决方案可能无法解决即时性的要求：如果某个任务需要每5分钟运行一次，由于Cron服务迁移导致的每次一分钟，或者两分钟的延迟可能是无法接受的。在这种情况下，热备任务，在主任务迁移的时候及时介入恢复服务可以极大降低系统的不可用时间。

对需求的扩展

单物理机系统通常会将所有运行着的程序聚合在一起，仅仅提供非常有限的资源隔离手段。虽然现在容器（container）技术已经很流行，但将机器上每个组件都隔离在自己的容器里一般是不常见的，也是不必要的。因此，如果Cron系统部署在单个物理机上，crond和其他的Cron任务经常是没有隔离的。

而将Cron系统部署到整个数据中心级别，一般来说意味着将进程部署到容器中，以便更好地进行资源隔离。在这里，隔离是很有必要的，因为数据中心中每个独立进程不会互相影响是一个基础性假设。为了保证这种假设有效，我们必须在每个进程运行之前知道它预计使用的资源—不管是Cron系统本身，还是该系统所要运行的任务。某个Cron任务可能由于数据中心没有足够资源满足该任务而延迟。这种资源要求，加上用户对任务运行情况的监控需要，使得我们需要记录Cron任务的全过程，从预计执行一直到任务终止。

将进程启动过程与具体运行的机器分离使得整个Cron系统需要处理“部分启动”的故障类型。Cron任务的多样性意味着某些任务在启动的时候可能需要发送多个RPC，有时我们会遇到某个RPC成功，而其他

RPC不成功的问题（例如，发送RPC的进程本身在发送过程中崩溃了）。Cron 故障恢复过程必须要处理这种情况。

在讨论故障模式的时候，一个数据中心相比单个物理机来说是一个更加丰富的生态系统。Cron服务，当大规模部署时从一个简单的二进制文件变成了一个具有许多明显和不明显的依赖的服务。对Cron这样的基础服务来说，必须保证在数据中心遇到部分故障的时候（例如部分供电系统故障，或者存储系统故障），整个服务仍然能够正常运行。通过在数据中心内部分散地同时运行多份调度器的副本，我们可以避免单个供电单元故障造成整个Cron系统不可用。

将Cron服务部署在全球范围内可能是可行的，但是将Cron部署在一个单独的数据中心中有其对应的好处：该服务与其对应的数据中心任务分发系统延迟很低，同时共享一个故障域。毕竟，数据中心任务分发系统是Cron服务的一个核心依赖。

Google Cron系统的构建过程

本节主要展示构建一个可靠的大型分布式Cron系统所必须要解决的问题。同时也强调了Google的分布式Cron服务的几个设计要点。

跟踪Cron任务的状态

如上一节所述，我们需要记录关于Cron任务的一些状态信息，并且必须能够在系统发生故障的时候快速恢复。更重要的是，该状态信息的一致性是关键！还记得前文所述的那些发送邮件和计算工资之类的非幂等的Cron任务吗？

跟踪任务的状态有两个选项：

- 将数据存储在一个可用度很高的外部分布式存储上。
- 系统内部自行存储一些（很小量的）状态信息。

当Google设计分布式Cron任务的时候，我们选择的是第二个选项。这样选择的原因有以下几个：

- 分布式文件系统，包括GFS和HDFS通常用来存储非常大的文件（例如，网页爬虫程序的输出文件），而我们需要存储的Cron任务状态信息通常来说是非常小的。小型写操作在分布式文件系统上的开销很高，同时延迟也很高，因为这些文件系统就不是为这种类型的写操作进行优化的。

- 基础服务，也就是那些失效时会带来许多副作用的（就像Cron这样的）服务应该依赖越少越好。即使数据中心的一部分出现故障，Cron服务也应该能够持续工作一段时间。但是这个要求并不一定意味着存储区域一定要在Cron进程内部（存储部分其实只是一个实现细节）。然而，Cron服务应该可以独立于下游系统而运行，以便服务更多的内部用户。

Paxos协议的使用

我们为Cron服务部署了多个副本，同时采用Paxos分布式共识算法保证它们状态的一致（参见第23章）。就算某些基础设施出现故障，只要整个组中大多数成员可用，整个分布式系统就可以顺利地进行状态变更。

如图24-1 所示，分布式Cron系统使用了一个单独的领头人任务，该副本是唯一一个可以修改共享状态的副本，也是唯一一个可以启动Cron任务的副本。我们基于Paxos协议的变种—Fast Paxos（参见文献[Lam06]）协议的优势，将领头人副本和Cron服务的领头人合为一个。

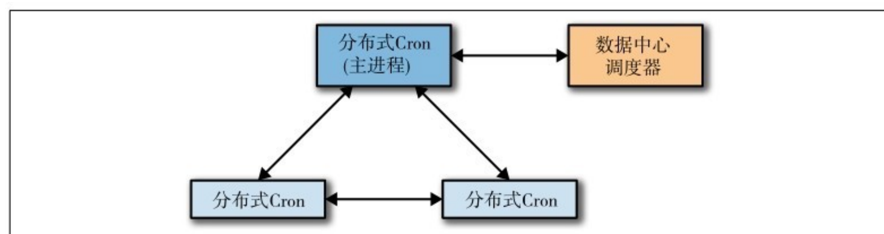


图24-1：不同分布式Cron副本之间的交互过程

如果领头人副本崩溃了，Paxos 小组的健康检查会很快发现这个问题（通常在几秒之内）。由于已经有另外几个Cron进程处于启动状态，所以可以很快地选举出一个新的领头人。一旦新的领头人被选举出来，

根据Cron服务自定义的选举过程使得新的领头人进程接手之前领头人未完成的工作。虽然Cron服务的领头人进程和Paxos协议的领头人进程其实是一个，但是Cron领头人在选举结束后需要完成一系列额外的操作。领头人过程的选举速度很快，使我们可以很轻松地保证在可接受的一分钟故障时间内进行故障切换。

各个副本利用Paxos协议同步的最重要的状态信息就是哪些Cron任务已经被启动了。该服务需要不停地以同步方式通知大多数的副本每个计划任务的启动和结束信息。

领头人角色和追随者角色

正如前文所述，Cron服务使用Paxos协议分配两个角色：领头人和追随者。下文会详述每种角色的职责。

领头人角色

领头人进程是唯一一个主动启动Cron任务的进程。该领头人进程内部有一个内置调度器，与本章开头提到的简单的crond实现非常类似。该调度器按照预定的启动时间排序维护一个Cron任务列表。领头人进程在第一个任务的预期执行时间之前一直处于等待状态。

当到达预定启动时间时，领头人宣布它将要开始启动该Cron任务，同时计算新的启动时间，和普通的crond实现一样。当然，与普通的crond类似，自从上次执行过后，Cron任务的启动条件可能已经变化了，这些启动信息也必须同步给其他所有的跟随者角色。简单地标记某个Cron任务是不够的，应该将某次启动时间作为唯一标识符标记；否则在记录Cron任务的启动时可能会产生歧义（这种歧义通常出现在那些高频任务中，比如那些每分钟运行一次的任务）。如图24-2 所示，全部通信都是基于Paxos协议之上完成的。

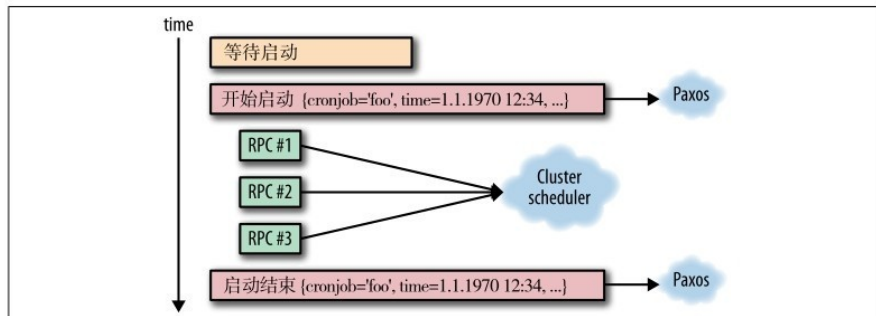


图24-2：以领头人视角画的Cron任务启动过程示意图

Paxos 通信的同步性是很重要的，这样Cron任务的实际启动在得到Paxos法定仲裁过程结束之前不会进行。Cron服务需要知道每个任务是否已经启动了，这样才能在领头人切换的时候决定正确的操作。如果这些操作是异步进行的，可能意味着整个任务的启动过程都在领头人进程上完成，而没有通知到其他的副本。当故障切换的时候，新的领头人副本可能会重新进行这次启动，因为它们不知道这个任务已经被启动过一次了。

Cron任务启动过程的完成也需要通过Paxos协议同步通知给其他的副本。要注意的是，我们这里仅仅记录了Cron服务在某个时间试图进行一次启动操作，并不关心这次启动是否真正成功或者是否由于外部原因失败了（例如，如果数据中心任务分发系统当前不可用）。下文会提到如何处理发生在这项操作过程中的问题。

另外一个作为领头人角色非常重要的功能是，当其由于各种原因失去领头人角色时，该进程必须立刻终止一切和数据中心任务分发系统之间的交互。领头人角色必须意味着对数据中心级别任务分发服务的独占性。如果缺少了这种独占性，旧的领头人进程可能会和新的领头人进程同时进行相互冲突的操作。

追随者角色

追随者副本需要持续跟踪领头人进程提供的目前的系统状态，以便在需要的时候及时替换。所有的状态改变都是从领头人角色基于Paxos协议传递的。就像领头人角色那样，跟踪者也必须要维护系统中所有

Cron任务的列表，这个列表必须在所有副本中保持一致（这就是要使用Paxos协议的原因）。

当接收到某个已经执行的“启动”通知时，追随者副本需要更新该副本内部该任务的下次预计启动时间。这项重要的状态改变（是同步进行的）保证了所有Cron任务在整个系统中的状态是一致的。我们同时也会记录所有的未完成启动。

如果领头人进程崩溃，或者由于其他某个原因不再正常工作（例如，由于网络分区问题无法联系到其他的副本），某个追随者进程将会被选举为新的领头人进程。该选举过程必须要在一分钟之内完成，这样可以避免大幅延迟或者跳过某个任务的执行。一旦一个领头人被选举出来，所有的未完成状态的启动过程必须被结束。这个过程可能是很复杂的，通常需要Cron系统和数据中心基础服务的共同协作才能完成。下面这一小节会详细讨论如何解决这种部分性失败情况。

解决部分性失败情况

正如上文所述，领头人进程和数据中心调度系统之间的单个任务启动的过程可能在多个RPC中间失败。我们的系统应该能处理这种状况。

前文说过，每个任务启动都具有两个同步点：

- 当启动执行之前
- 当启动执行之后

这两个同步点使我们可以标记每一次具体的启动。就算启动过程仅仅需要一个RPC，我们怎么能够知道这个RPC是否已经发出了呢？这里还包括在标记启动已经执行之后，但是启动结束通知发送之前领头人进程就崩溃了的情况。

为了能够判断RPC是否成功发送，下列情况中的一个必须被满足：

- 所有需要在选举过后继续的，对外部系统的操作必须是幂等的（这样我们可以在选举过后重新进行该操作）。

- 必须能够通过查询外部系统状态来无疑义地决定某项操作是否已经成功。

这两个选项中的任意一个都是很大的挑战，可能很难实现。但是实现其中的至少一个是在大型分布式系统中应对单点或多点故障并准确运行一个Cron系统的前提。如果没有正确地处理这些情况，可能会导致错过某些任务的执行，或者重复运行某些任务。

在数据中心中管理逻辑性任务的大部分基础设施（例如Mesos）会为这些任务提供某种命名服务，使我们可以通过这些名字查找任务状态、停止任务，或者进行其他的维护操作。一种解决幂等性问题的方案是提前构建一致的任务名称，并且分发给所有的Cron服务副本（这样就可避免在数据中心调度器上进行任何的修改操作）。如果领头人副本在启动过程中崩溃，新的领头人副本可以通过预计算的任务名称来查询任务的状态。

这里要注意的是，就像在Cron内部使用名字和预计启动时间唯一标识某个任务那样，我们也需要在数据中心调度器上的任务名称上包含该次启动时间（或者用某种其他可获取的方式标记这个信息）。在常规操作中，Cron服务应该可以很快地完成故障迁移，但是也会有意外情况发生。

上文说过，我们在副本之间同步信息的时候会记录预期启动时间。同样的，我们需要将Cron服务与数据中心调度器的交互也使用预期启动时间唯一标识。例如，假设有一个高频运行的，但是运行时间很短的任务。该任务成功启动，但是在启动成功的信息同步到其他副本之前，领头人进程崩溃了。同时故障切换机制由于某种问题造成了延迟，迟到足够这个任务已经结束了。新的领头人副本查询该任务的状态，发现该任务目前处于完成状态，于是尝试再次启动该任务。如果预期启动时间被包含在任务名中，新的领头人副本就可以准确地知道这个任务其实是任务的某个特定启动，于是就避免了这种双重启动情况的发生。

在实际的实现中，状态的查询由一个比较复杂的系统组成，这主要是由于底层基础设施的实现细节所决定的。然而，前面的描述适用于任何系统。取决于具体的基础设施，实现者可能要在双重启动的风险和错

过启动的风险中进行抉择。

保存状态

使用Paxos 协议来达成共识只是状态问题的一部分。Paxos 基本上是一个只能新增的日志，在每次状态变化后同步地新增。Paxos协议的这种特性意味着以下两个问题：

- 日志需要定期压缩，以防无限增长。
- 日志必须要存储在某个地方。

为了避免Paxos日志的无限增长，可以简单地将目前的状态进行一次快照（snapshot）。这就意味着我们可以不再需要重新回放之前的所有状态改变来得到目前的状态。提供一个例子：如果我们的状态变化存储在日志中的时候是“将某个计数器加1”，那么在1000次迭代之后，可以用一个“将计数器设置为1000”来替代1000条记录。

在日志丢失的情况下，我们只会丢失上次快照之后的信息。快照也是我们最重要的状态信息—如果丢失快照信息，我们就必须从零开始，因为已经丢失了全部内部状态。相比之下，丢失日志仅仅会将Cron系统重置回上次快照的时间。

存储数据有两个方案：

- 外部可用的分布式存储。
- 在系统内部存储少量的数据。

当设计该系统时，我们采取了一个综合两个选项的方案。

我们将Paxos日志存储在服务副本运行的本地磁盘上。在标准情况下，我们有三个副本运行，也就是有三份日志。同时我们也将快照信息保存在本地磁盘上，然而因为这些信息很重要，会同时将它们备份到一个分布式存储上去。这样可以为三个机器同时出现问题的情况提供保护。

我们并不将日志直接存储在分布式文件系统上。我们认为丢失日志，也就是丢失一小部分最近的状态改变是一个可以接受的危险。将日志保存在分布式文件系统上可能会造成很严重的性能问题—由大量的小型写操作引起。三个机器同时出现故障是很罕见的，而且如果的确同时发生故障了，我们会自动从快照中恢复。于是，我们仅仅只会丢失一小部分日志信息：这些都是自从上次快照之后产生的。而快照是根据预先配置的时间间隔自动产生的。当然，这些妥协可能根据基础设施的细节而不同，也与Cron系统本身的要求有关系。

除了在本地存储的日志和快照，以及分布式文件系统上的快照备份之外，一个刚刚启动的副本可以从另外一个已经运行的副本上获得最新的快照和日志。这个能力使得启动过程可以独立于本地物理机器的任何状态。因此，将系统的某个副本重启，重新调度到另外一台物理机器上（或者由于物理机器故障原因迁移）对服务稳定性来说没有影响。

运维大型Cron系统

运维一个大型Cron系统同样有一些较小但是也很有趣的问题。传统的Cron系统规模通常很小，最多可能由几十个任务组成。但是，如果我们在一个数千台机器的数据中心上运行Cron系统，系统的用量可能会大幅增加，同时可能会造成一定的问题。

一定要小心任何大型分布式系统都熟知的问题：惊群效应（thundering herd）。根据用户的配置，Cron服务可能会造成数据中心用量的大幅增加。当人们想要配置一个“每日任务”时，他们通常会配置该任务在午夜时运行。这个配置可能在一台机器上还可行，但是当你的任务会产生数千个MapReduce工作进程的时候就不可行了。尤其是当其他30个团队也按照同样的配置在同一个数据中心来运行每日任务呢？为了解决这个问题，我们在crontab 格式上增加了一些扩展。

在普通crontab格式中，用户用分钟、小时、月中的日期（或者周内的日期），以及月份来标记某个任务应该运行的时间，或者使用“*”来标记任意值。每天在午夜时候运行会将crontab标记为“0 0 * * *”（也就是0分0时，月内的每一天，每个月，以及周内的每一天都会运行）。我们在这里增加了一个问号“?”，意味着任意一个值都可以接受，Cron系统可以根据需要自行选择一个时间。用户可以将任务分散在整个事件序列里（例如 `0..23` 中的某个小时），因此可以将这些任务分散得更为均匀。

就算引入了这个改动，Cron任务带来的系统负载仍然是非常尖锐的。图24-3所示的是，聚合过的Google全球Cron任务启动数量图。这个图片集中展示了Cron任务经常性的负载峰值，这是因为某些任务需要在特定时间启动——例如，由于对外部事件的某些临时依赖造成。

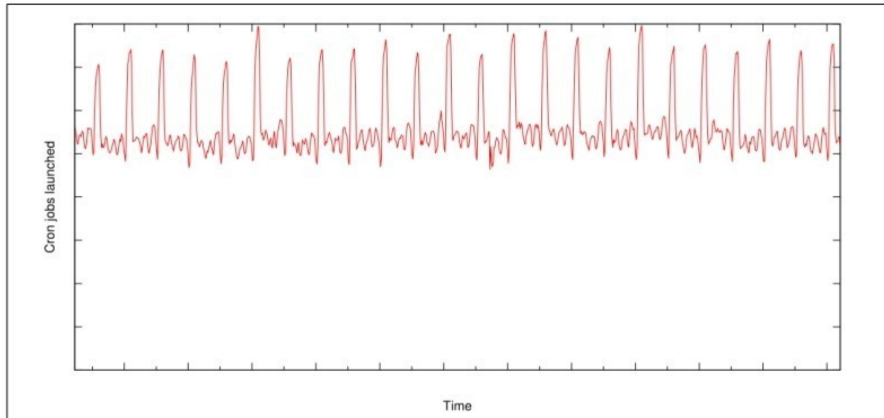


图24-3：全球Cron任务的启动数量

小结

Cron服务是UNIX系统十多年来的一个基础服务。随着行业向大型分布式系统迁移，数据中心逐渐成为事实上最小的硬件单元，也就意味着技术栈中很多部分都要改变。Cron也不例外。Google的新设计是建立在对该服务的必要属性以及相关任务的具体需求之上的。

我们讨论了分布式系统环境下出现的新约束条件，以及一个基于Google解决方案的新的系统设计。这个解决方案需要分布式环境下的强一致性保障。因此，分布式Cron系统的核心是Paxos协议，一个在分布式不可靠系统达成一致的算法。Paxos协议的使用，以及对大型、分布式环境中任务失败模式的分析使得Google可以构建出一个非常可靠的Cron服务。

第25章 数据处理流水线

作者：**Dan Dennison**

编辑：**Tim Harvey**

本章关注数据处理流水线在实际应用中面临的复杂问题。这里讨论了周期性运行的数据流水线与持续运行永不停止的数据流水线面临的共

同问题，也讨论了它们的不同点，这往往是运维的痛点所在。本章提出了一个崭新的、更可靠的、更易扩展的领头人-追随者模式，可以用来替代处理海量数据的周期性流水线模型。

流水线设计模式的起源

经典的数据处理方式是通过一个程序读取输入，执行某种模式变换，然后输出新的数据。一般来说，这种程序由类似于Cron的某个周期性的调度程序控制。这种设计模式被称为数据流水线（data pipeline）。数据流水线是一个由来已久的概念，与协程（coroutine，参见文献[Con63]）、DTSS通信文件（参见文献[Bul80]）和UNIX管道（参见文献[McI86]），以及后来的 ETL 管道[81]类似。但是数据流水线随着“大数据”或者“传统数据处理应用无法处理的特别巨大或者超级复杂的数据集”逐渐增多而流行起来。[82]

简单流水线设计模式与大数据

对大数据进行周期性的或者是持续性的变形操作的程序通常被称为“简单的，单相流水线”（simple,one-phase pipeline）。

由于大数据与生俱来的海量级别和处理的高复杂度，这种程序通常会被串联起来执行，一个程序的输出作为另外一个程序的输入。这种编排方式有很多种原因，但是最常见的原因是，这样设计有助于理解系统的逻辑，但是这在系统效率方面并不一定是最优方案。我们将这样构建的程序称为多相流水线（multiphase pipeline），因为整个链条中的每一个程序都是一个独立的数据处理单元。

一个数据流水线中串联的程序数量多少称为该流水线的深度（depth）。因此，一个很“浅”的流水线可能只包含一个程序（深度为1），而一个很“深”的流水线可能包含数十个，甚至几百个程序。

周期性流水线模式的挑战

周期性流水线在工作进程数量可以满足数据量的要求，以及计算容量足够执行需要时，相对比较稳定。而且，我们可以通过保证在流水线

中串联的程序数量相对固定以及程序之间的吞吐量基本一致的方法来排除整个流水线中的瓶颈及其他不稳定因素。

周期性的数据流水线非常实用，Google也经常使用，为此还编写了一系列程序框架，例如MapReduce（参见文献[Dea04]）和Flume（参见文献[Cha10]）等。

然而，SRE运维经验证明，周期性的数据流水线模型是非常脆弱易坏的。我们发现当第一次安装周期性流水线时，工作进程数量、运行周期、分块处理技术，以及其他参数仔细调校过后，整个流水线的性能可能很稳定。但是随着数据量的自然增长等种种变化会对整个系统造成压力，导致各种各样的问题出现。这种问题包括任务运行超时、资源耗尽，某些分块处理卡住导致整体运维压力上升等。

工作分发不均造成的问题

处理大数据的一个关键思想是利用“令人羞愧的并发性”（embarrassingly parallel）算法（参见文献[Mol86]）将巨大的工作集切割为一个个可以装载在单独的物理机器上的小块。有的时候工作分块所需要的处理资源是不等的，而且理解为什么这一块工作需要更多的资源是很困难的。举例来说，在一个按客户分块的工作集中，某些客户的工作分块可能会比其他的都大，由于单个客户已经是系统的最小分块单元，整个数据流水线的运行时间就至少等于处理最大的客户的工作分块所需的时间。

“卡住的工作分块”问题可能是由于资源在集群中分配不均衡，或者是资源分配过量导致的。这个问题也可能是由于某些实时操作在流式数据上实现的困难性导致的，例如“排序流式传输的数据”。这种类型的用户程序通常需要等待某个阶段的计算全部完成之后再进行下一阶段的计算，而“排序”需要等待全部数据就位才能继续。这样有可能会极大地拖慢整个数据流水线的完成时间，因为完成整个流水线要依赖于整个流水线中性能最差的分块的完成。

如果这个问题被工程师或者集群监控系统发现了，那么他们采取的措施可能会使得事情变得更糟。例如，“默认的”或者“合理的”针对卡住

的工作分块问题的操作可能是将整个任务杀掉，使整个任务重启，因为可能是某种非决定性的意外因素导致的任务卡住。然而因为整个流水线的实现中并没有包含状态记录，所有的工作分块都需要重新进行计算，这样前一次运行消耗的时间、CPU以及人力成本就全都浪费了。

分布式环境中周期性数据流水线的缺点

处理大数据的周期性数据流水线在Google使用非常广泛，所以Google的集群管理解决方案里包括了针对这种数据流水线的另外一套调度机制。由于周期性任务不像持续运行的流水线任务那样，它们经常以低优先级批处理任务方式运行，针对它们单独进行调度是很有必要的。将这种任务定位为低优先级是合理的，因为流水线任务通常不像面向用户业务那样对延迟敏感。而且，为了降低成本、最大化机器负载，Borg系统（Google的集群管理系统，参见文献[Ver15]）将批处理任务指派给可用的机器处理。但是，这种低优先级可能会造成任务启动很慢。

利用这种机制运行的任务面临着一系列天然的限制，这使得它们和普通任务截然不同。例如，这些任务依赖那些大型面向用户的高优先级任务之间的资源缝隙运行，低优先级的资源、价格、稳定性以及可用性都没有保障。执行成本与启动延迟成反比，与消耗的资源成正比。虽然实践中这种批处理调度方式工作得还可以，但是如果在集群压力很大的时候，超量使用批处理调度器（参见第24章）容易使任务经常被其他任务所挤走（preemption）（见文献[Ver15]的第2.5节）。在这种妥协作用下，运行一个良好调校过的周期性数据流水线实际上是在资源成本与挤占风险中的一个微妙平衡。

对于一个每天运行一次的流水线来说，几个小时的启动延迟可能是可以接受的。但是随着执行频率的增高，每次执行之间的最小时间很快会与平均启动延迟持平，这就使得周期性数据流水线的执行延迟有了一个下限。超过这个阈值继续减少任务执行的间隔周期只会造成更多的问题而不能让任务执行得更快。实际造成的问题和所使用的批处理调度策略有关。例如，新的执行可能会在集群调度器上累积，因为前一次运行还没有结束。更糟糕的是，正在执行的、马上就要结束的运行可能会由于下一次执行马上就要开始而被杀掉，加速运行的操作反而会造成整个

流水线完全停滞。

请注意图25-1所示的下滑的执行时间线与调度延迟的交叉。在这个场景中，将这个需要大约20分钟完成的任务的执行周期降低到40分钟以下时可能会造成重叠执行，这可能会造成异常情况。

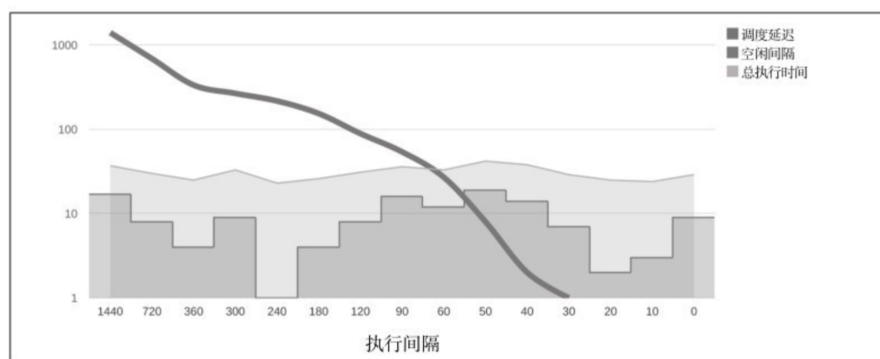


图25-1：周期性流水线执行间隔时间与等待时间（对数级Y轴）

这个问题的解决方案是为正常运行提供足够的服务容量。然而，在一个共享的、分布式的环境中资源的获取要依赖供需关系的变化。正如我们所预计的那样，开发团队一般不愿意费力获取资源，再将其贡献给一个公共池中共享。为了解决这个问题，批处理调度资源和生产系统调度资源必须有所区分，以便更好地区分资源的获取成本。

监控周期性流水线的问题

对那些有充裕时间运行的流水线来说，整个流水线的全局运行指标与运行时性能指标的实时监控都很重要。这是因为，实时监控可以提供更多的运维支持，包括提供紧急情况下的响应帮助。在实践中，标准的监控模型在任务执行过程中不停地收集指标，仅在完成时才一次性提交。如果任务在执行过程中失败，就不会有统计信息。

持久性的流水线不存在这样的问题，因为它们的任务是一直运行的，而它们的指标信息被设计为一直可用。周期性的流水线并不一定都具有监控难度问题，但是我们在实践中发现两者强烈相关。

惊群效应

在执行问题和监控问题之外，还有分布式系统常见的“惊群效应”，我们在第24章中讨论过这个问题。对一个足够大的周期性流水线来说，每一次运行，可能有几千个工作进程立即启动。如果一个服务器上有太多工作进程，或者这些工作进程配置错误，并用错误的方法重试等，可能会导致所运行的物理服务器过载，甚至于分布式集群服务也会过载，网络基础设施等也会出现问题。

使这个情况变得更糟的是，如果没有实现重试逻辑，在任务失败时，之前所做的工作可能会丢失，造成正确性问题。如果实现了重试逻辑，但是重试逻辑过于简单，没有被仔细调优，在重试的时候可能会放大问题。

人工干预也有可能使问题变得更严重。经验不足的工程师在流水线无法按时完成的时候经常会向其中添加更多的工作进程，使得整个问题变得更糟。

不管“惊群效应”的根源在哪里，没有什么比一个问题不断、由10,000个工作进程组成的流水线任务对集群内的服务造成的压力更大的了。

摩尔负载模式

有的时候这种“惊群效应”不那么容易发现。另外一个相关问题，我们称之为“摩尔负载模式”（Moiré Load Pattern）。该问题是指两个或者更多的流水线任务同时运行时，某些执行过程重叠，导致它们同时消耗某个共享资源。这个问题也会在持续性流水线中出现，但是在负载比较平衡的时候就很少出现了。

摩尔负载模式在流水线共享资源使用图表中表现得非常明显。例如，图25-2标记出了三个周期性流水线对资源的使用图。在图25-3中，将之前的图表转化为层叠方式可以看出当聚合负载接近1.2M的时候将会给on-call工程师造成麻烦。

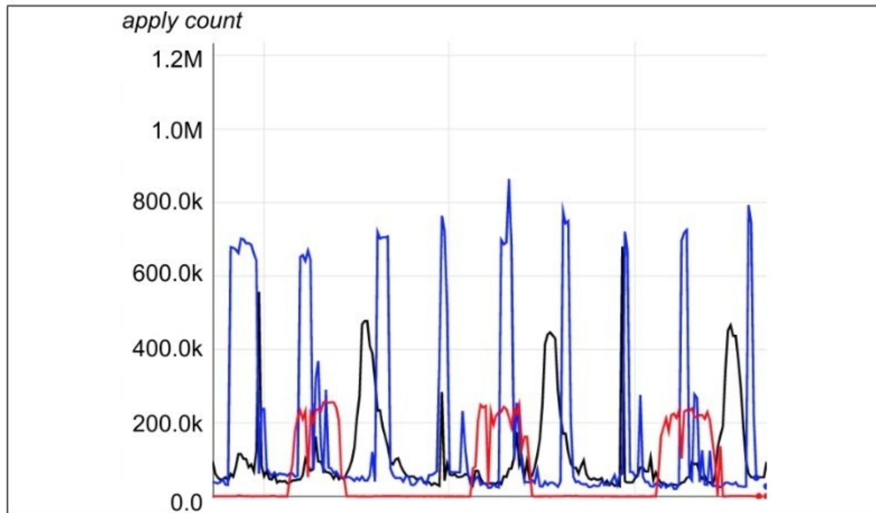


图25-2：不同的基础设施中的摩尔负载模式

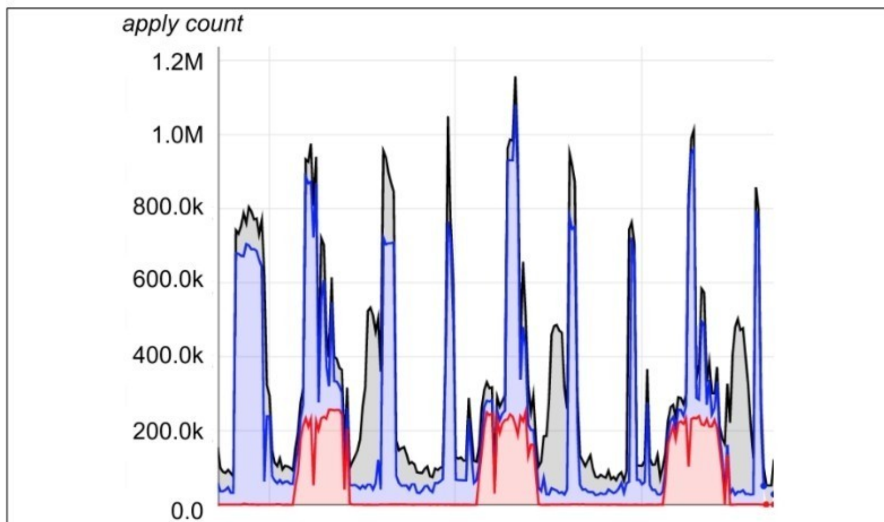


图25-3 ：在共享基础设施中的摩尔负载模式

Google Workflow简介

当一个一次性批处理流水线由于业务需要而持续不断地需要更新结果时，流水线开发小组要么考虑将原始设计重构以满足目前的需求，或者将业务迁移到一个持续性流水线模式上。不幸的是，业务需求常常在最不适合重构的时间出现。更新、更大的客户在开发团队面临着严峻的扩展性问题的时候经常还要增加新的功能，并且要求这些需求都在不可

改变的最后期限之前完成。为了战胜这些挑战，在系统设计阶段提议一个数据流水线的时候必须要注意以下几个细节：首先要规划出预期的增长轨迹，[83]预计对设计更改多少，预计需要多少额外资源，以及预计业务的延迟性需求。

针对这些需求，Google在2003年开发了一个名为“Workflow”的系统，使得大规模持续数据处理成为可能。Workflow使用领头人-追随者（工作进程）分布式系统设计模式（参见文献[Sha00]），以及“流式系统”（system prevalence）设计模式。[84]这样的组合可以实现超大规模的交易性数据流，同时可以提供保障“仅运行一次”的语义来保障正确性。

Workflow 是模型—视图—控制器（MVC）模式

根据流式系统的工作原理，Workflow可以被认为是分布式系统中与用户界面设计中的MVC模式相同的一种模式。[85]如图25-4所示，这种设计模式将某个软件程序分为三个相连的组件，来分隔内部信息、信息的展示和信息的输入部分。[86]

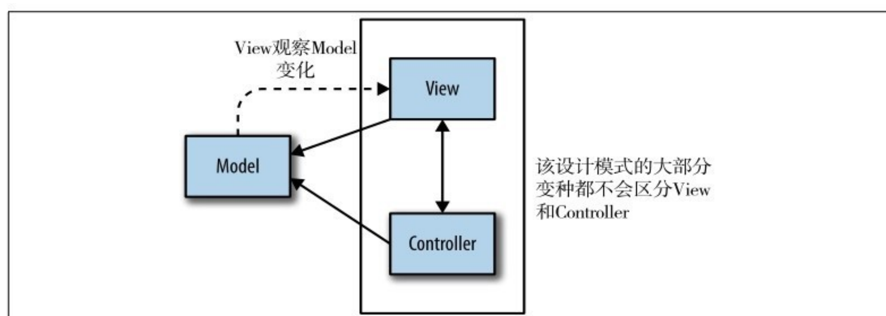


图25-4：用户界面设计中常见的MVC模式

Workflow采用了这个模式，模型（Model）由一个被称为“主任务”的进程持有。主任务使用流式系统模式将所有的任务状态保存在内存中，同时同步地将每一次修改以日志方式记录在持久化磁盘上。视图（View）是那些作为整个流水线子组件，不断向主任务更新它们所见的系统状态的工作进程。虽然整个流水线的全部数据可以保存在主任务中，但是为了获取更好的系统性能，主任务通常只会保存具体工作的指针，而真正的输入和输出数据会保存在一个常见的文件系统或者其他存

储系统中。根据这个比喻，工作进程是完全无状态的，可以在任一时间被抛弃。控制器（Controller）是可选的，可以加入进来支持一系列的辅助活动，比如对流水线的实时伸缩，对状态的快照，以及工作周期的管理，回滚流水线的状态，甚至是在紧急时刻负责停止一切系统行为。图25-5 描述了这种设计模式。

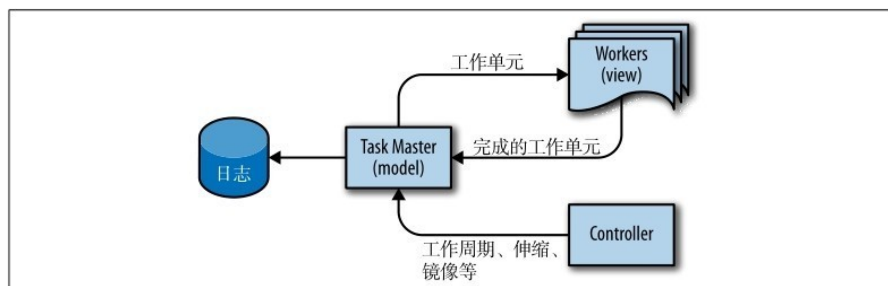


图25-5: Google Workflow系统采用的MVC模式

Workflow中的执行阶段

我们可以通过将工作进程进一步划分为更小的任务组，而将流水线的“深度”随意增加。每个任务组负责处理该执行阶段的数据，可以对某一小块数据进行任意操作。于是，我们可以在任何一个阶段很容易地实现 Mapping、Shuffling、排序、分割以及合并等操作。

某一个阶段通常和几种工作进程类型关联在一起。某个类型的工作进程可能有多个并行实例，工作进程也可以通过选择不同类型的工作来执行以进行自我调度。

工作进程处理前序阶段产生的工作单元，同时生产出新的输出单元。输出可以终止整个执行，也可以作为其他处理阶段的输入。在系统中，我们可以很容易地保证所有的工作都被精确地执行了一次且仅仅一次（至少是那些记录在永久状态中的工作）。

Workflow正确性保障

将整个流水线的每一点细节都保存在“主任务”进程中是不太可行的，因为主进程还要受到内存尺寸的限制。然而我们仍然提供双重保障，因为在主任务的状态中给所有数据分配了一个唯一命名的指针，同

时每个工作单元都有一个全局唯一的租约。工作进程通过租约来获取工作，而且仅仅允许那些目前持有有效租约的进程汇报状态。

为了避免在某些情况下一个不受管理的工作进程擅自处理某个工作单元，与目前分配的进程造成冲突，每一个工作进程打开的输出文件都有一个全局唯一的名字。这样，脱离管理的工作进程也可以继续写入输出文件，它们试图将文件提交回主进程的时候，主进程将会拒绝它们的请求，因为目前有另外一个工作进程正在持有这段租约。更重要的是，这样一来脱离管理的工作进程无法影响到其他的正常进程，因为每个工作进程都有一个全局唯一的文件名。这样，我们就可以提供正确性的双重保障：输出文件永远是全局唯一的，而流水线状态永远只能由那些拥有租约的任务更新。

如果认为这种双重保障还是不够，Workflow系统同时会记录所有任务的版本信息。如果任务更新了，或者任务的租约发生了变化，那么这种操作将会产生出一个新的任务，替换掉之前的那个，同时带有一个新的全局ID。因为所有的流水线配置和工作进程信息都存储在主任务中，一个工作进程必须同时持有一个活跃的租约，并且拥有配置文件中的正确ID。如果配置文件在工作进程工作的时候更新了，那么所有的工作进程虽然仍然拥有租约，也无法提交状态。因此，在配置文件更新之后，所有的工作成果都会和新的配置文件保持一致，哪怕这样会使得之前的一些工作进程的工作被抛弃了。

这样，Workflow就提供了三重保障：配置文件、租约和唯一文件名。但是，有的时候这样还是不够的。

例如，如果“主任务”的网络地址改变了，另外一个“主任务”运行在同样的地址上替代了之前的那个会怎么样呢？如果某个内存错误影响了IP和端口地址导致RPC发给了另外一个错误的“主任务”怎么办？或者更常见的，如果某个人错误地在一组主任务前面加入了一个负载均衡器呢？

Workflow协议在每个任务的元数据中嵌入了一个服务器令牌，用来唯一标记某个特定的“主任务”。这样可以避免某个异常的或者是配置错误的“主任务”破坏整个流水线的状态。客户端和服务端在每个操作时都

会检查令牌，这样就避免了在配置错误的时候难以查找错误。

综上所述，Workflow为正确性提供了4点保障：

- 配置文件本身作为屏障，这样可以保障工作进程的输出永远与配置一致。
- 所有的工作结果都必须由当前拥有租约的进程提交。
- 工作进程的输出文件都是全局唯一命名的。
- 客户端和服务器会在每次操作的时候校验“主任务”的令牌。

读到这里，可能你觉得可以抛弃这种特制的“主任务”而直接使用Spanner（参见文献[Cor12]）或者另外一个数据库来实现。然而Workflow系统特殊的地方在于每个任务都是独特的、不可变的。这样的两个特性使得Workflow系统免于许多种大型任务分发系统面临的困难。

例如，工作进程租约的获取是任务的一部分，每个租约的改变都需要一个新的任务。如果直接使用一个数据库，用它的交易日志作为我们的日志，那么每个读操作都必须成为一个长期运行的交易操作的一部分。这肯定是可行的，只不过会非常低效。

保障业务的持续性

大数据处理流水线需要在各种失败条件下持续运行，包括光纤被切、天气状况、连锁性供电故障等，这些故障可以使整个数据中心不可用。在这种情况下，没有采用流式设计来提供强任务状态保障的流水线经常会进入一个未定义的状态。这个架构上的问题使得业务持续性非常难以达到，有可能需要消耗大量的人力物力来恢复流水线的运行和其中的数据。

Workflow通过持续性运行的流水线完美地解决了这个难题。“主任务”通过将日志存储在Spanner上，利用其全球可用、全球一致的特性作为一个低吞吐量的文件系统。为了决定哪个“主任务”可以写入数据，每个“主任务”使用分布式锁服务Chubby（参见文献[bur06]）来选举一个

领头人，同时将结果保存在Spanner中。最后，客户端使用内部的一个名称系统来查询目前的“主任务”。

因为Spanner并不是一个高吞吐量的文件系统，全球分布的Workflow系统使用一种“任务引用”和两个或多个本地运行在不同数据中心的Workflow系统来运行任务。随着任务单元经过流水线时，相对应的任务引用会被插入到全球Workflow系统中，带有标记“stage　1”，如图25-6所示。随着任务结束，任务引用将会从全球Workflow系统中删除，如图25-6中的“stage　n”。如果无法从全局Workflow系统中删除任务引用，本地Workflow系统将会暂时停止，直到全球Workflow系统再次可用，以保障交易的正确性。

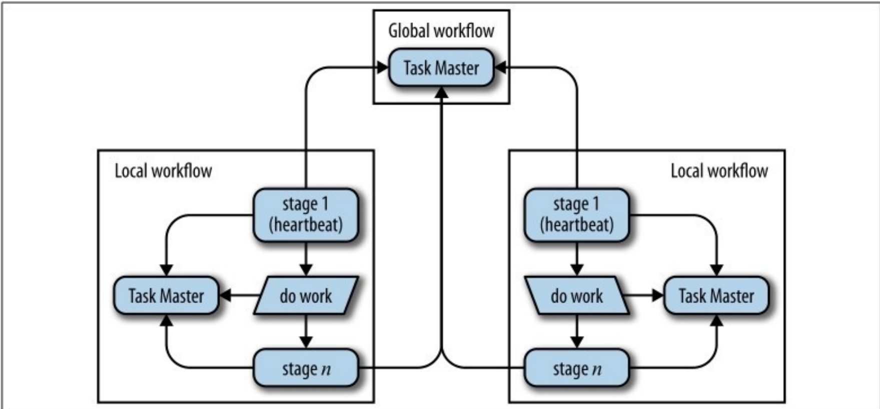


图25-6：使用Workflow流水线的分布式数据处理流示例

为了做到自动灾难迁移，图25-6中的一个助手二进制文件（标签为stage　1）会在每个本地Workflow系统中运行。本地Workflow系统除此之外就没有其他的改动了，就像图中“do　work”那个盒子中那样。这个助手二进制文件相当于MVC中的控制器角色，负责创建和删除任务引用，同时负责在全球Workflow系统中更新一个特殊的心跳任务。如果心跳任务一段时间没有更新，另外一个远端的Workflow助手任务会将这个目前的任务接管过来，以保障系统正常运行，不受任何环境变化影响。

小结

周期性的数据流水线是很有价值的。但是如果一个数据处理问题本

身是持续性的，或者会自然增长成为持续性的，那么就不要采用周期性的设计方式，而是采用一种类似Workflow的设计特点的系统。

我们发现带有强一致性的持续数据处理系统，就像Workflow这样，在分布式集群环境中工作性能和扩展性都非常好。这样一个系统能够周期性地提供用户可以依赖的结果，并且是一个非常可靠且稳定的可运维系统。

第26章 数据完整性：读写一致

作者：**Raymond Blum、Rhandeev Singh**

编辑：**Betsy Beyer**

什么是“数据完整性（data integrity）”？当我们讨论一个面向用户的服务时，一切要以用户为准。

我们可以说数据完整性是指数据存储为了提供一个合理的服务质量，在可访问性和准确性方面必须达到的一个度量标准。但是这种定义是不全面的。

例如，如果用户界面上的一个Bug导致Gmail一直显示收件箱为空，用户可能认为数据已经丢失了。虽然数据并没有真正丢失，仅仅是无法访问，Google保管数据的能力还是会因此受到质疑，云计算的可信性也会受到影响。当Gmail出现错误提示或者是维护信息的时间过长，哪怕我们只是在“修复一点点元数据”,Google在用户心目中的信任度也会受到影响。

那么，无法访问的时间标准是什么呢？就像2011年Gmail事故展示的那样（参考文献[Hic11]）,4天已经足够久了—可能太久了！因此，我们一般认为“24小时”是针对Google App的一个合理时间阈值。

同样的逻辑也适用于Google Photos、Drive、云存储、云数据库等产品。因为用户有的时候并不会特意区分这些具体的产品，他们的逻辑

通常是“这个产品不就是Google的吗？”、“Google、Amazon 有什么区别：这个产品不都是云计算产品吗”。数据丢失、损坏、长时间无法访问对用户来说通常是一样的。因此，数据完整性适用于全部服务的所有类型的数据。当我们讨论数据完整性的时候，最重要的就是云服务依然对用户可用。用户对数据的可访问性是最重要的。

数据完整性的强需求

当考量一个系统的可靠性要求时，系统的“在线时间”（可用性）可能看起来会比数据完整性要求更严格。例如，用户可能无法忍受一个小时的E-mail服务宕机时间，但是他们可能可以接受等待四天时间（肯定会有很多怨言）来恢复自己的历史邮件。然而，我们有另外一种更好的方法来比较数据完整性与系统可用性。

如果某个服务的在线时间SLO是99.99%，那么一整年只能宕机1个小时。这个SLO 的标准其实很高，很有可能会超出绝大多数互联网用户，甚至是企业用户的预期。

但是，假设我们针对一个2GB的用户数据提供99.99% 的数据完整性SLO（也就是有200KB会被损坏），不管它们是文档、可执行文件，还是数据库。这么小的损失带来的问题大多数情况下将会是毁灭性的——可执行文件会乱码，数据库会彻底无法加载。

这样，从用户的视角来看，即使没有明确写出，每个服务都有独立的在线时间和数据完整性要求。在数据丢失之后才想起来和用户争论对数据完整性的定义是没有意义的！



那么，将上文的数据完整性定义重新修改一下之后成为，数据完整性意味着用户可以维持对云服务的访问，用户对数据的访问能力是最重要的，所以这种访问能力的完整性非常重要。

现在，假设某个用户数据每年会损坏或者丢失一次。如果这种丢失是无法恢复的，那么对这块用户数据来说，这块数据今年的在线时间就变成0了。避免这种灾难情况发生的最好手段就是主动探测加快速修复。

假设在另外一个平行宇宙中，每一次数据的损坏都能在用户被影响之前立即被发现，自动被删除、修复，最终在30分钟之内恢复访问。那么可以说针对这块用户数据，当年的可用率是 99.99%。

但是，神奇的是，从用户的角度来看，这个场景中的该块数据在可访问生命周期内的完整性仍然是100%（或者非常接近100%）。这个例子恰当地展示了保障超高数据完整性的手段是主动探测和快速修复能力。

提供超高的数据完整性的策略

针对快速检测及恢复丢失数据有很多可用的策略。所有这些策略都是在针对用户可见的数据在线时间以及数据完整性两者之间取舍。有些策略效果更好，有些策略则需要更多、更复杂的工程力量的投入。在这么多可选项中如何取舍？答案要从我们的服务的定位中寻找。

大多数云计算应用都是优化以下5项的某种组合：在线时间、延迟、规模、创新速度和隐私。以下是这5个名词的详细定义。

在线时间

经常也用“可用率（availability）”指代，代表着某个服务可以被用户使用的时间比率。

延迟

服务对用户的响应时间。

规模

某个服务的用户数量，以及能够维持正常服务水平的最高负载。

创新速度

某个服务能够在合理成本下，为用户提供更好的服务的创新速度。

隐私

这个名词的定义比较复杂。简单来说，本章将隐私的定义限制为仅仅针对数据删除：用户删掉服务中的数据后，数据必须在合理时间内被真正摧毁。

很多云应用程序都是在基于ACID[87]和BASE[88]某种组合的API上不停地演变来满足上述5个要求的。[89]BASE 相比ACID来说，使用更软性的分布式一致性要求来换取更高的可用性。BASE仅仅保障在某一个数据停止更新之后，在不同的数据存储位置（可能是分布各地的）中达到“最终一致”。

下面这个例子，描述了上述5个方面是如何在实际中互相影响的。

当创新速度压倒一切其他需求时，最终的应用程序肯定会依赖于开发者最熟悉的一系列API，而不一定是最优的。

例如，某个应用程序可能会利用某种二进制对象存储API（BLOB）[90]，如Blobstore，这其实是强调高负载下的可扩展性、高在线率、低延迟和低成本而忽略了分布式一致性的要求。为了在一致性方面进行补救：

- 该应用程序必须保证一小块、权威性的元数据存放在某种延迟较高、在线率稍低、成本更高的服务中，例如基于Paxos协议的Megastore中，可参见文献[Bak11]和[Lam98]。

- 该应用的某些客户端必须将一部分元数据信息缓存在本地，以便直接访问二进制对象，从而降低用户可见的延迟。

- 其他应用程序可能会简单地将元数据存放在Bigtable中，仅仅是因为应用程序开发者对Bigtable更熟悉，而没有考虑到Bigtable在分布式强一致性上的弱点。

这种云应用程序会在运行时面临各种各样的数据完整性挑战，例如，不同的数据存储之间的“引用完整性”（referential integrity），如上述例子中的Blobstore、Megastore以及客户端缓存三者之间的一致性。为了追求更快的创新速度，数据结构的改变、数据迁移、功能的不断累积、与其他应用程序整合点的不断增多，最后一定会产生一个极为复杂的环境，没有一个工程师能够理解其中各块数据之间的关系。

为了避免应用程序的数据出现问题，影响到最终用户，一个带外系统、专门负责检查和平衡各种数据存储的系统就很有必要了。本章后面的“第三层：早期预警”一节详细讨论了一个这样的系统。

除此之外，如果这样一个应用程序依赖的数据存储的备份是独立进行的，而不是彼此协调的（如在前述例子中的Blobstore和Megastore），那么各个数据存储备份之间存在的复杂数据关系会使数据恢复变得无比复杂。如上述这个例子，该应用程序进行数据恢复时，必须将复原的二进制对象与线上Megastore的内容进行对比，或者将恢复的Megastore 内容与可访问的二进制对象进行对比，或者将复原的二进制对象与复原的Megastore内容进行对比，这中间还需要考虑到客户端缓存的数据。

由于这种依赖关系和复杂交互的存在，我们应该向数据完整性投入多少精力呢？ 具体向什么地方投入才合适呢？

备份与存档

一般来说，公司会采用某种备份策略来“预防”数据丢失。然而，真正应该被关注的重点其实是数据恢复，这是区分备份与存档的重要区别。就像一句流行语说的那样：没有人真的想要备份数据，他们只想恢复数据。

那么，如何评判你的备份是不是只是一个存档，而不是灾难恢复中的可用的“备份”呢？



备份与存档最重要的区别就是，备份是可以直接被应用程序重新加载的。因此备份和存档的使用场景非常不同。

存档的目的是将数据长时间安全保存，以满足审核、取证和合规要求。在这些情况下的数据恢复通常不需要满足服务的在线率要求。例如，我们可能需要将财务交易数据保存七年时间。为了达到这个目标，我们可以每个月将累积的审核日志迁移到离线的、异地的长期存档存储系统上。读取和恢复这样的信息可能需要一周时间，在长达一个月的财务审核过程中，这是可以接受的。

相对存档，当灾难来临的时候，数据必须从真实的备份中快速恢复，最好能维持服务在线率的要求。否则的话，受影响的用户将由于数据问题无法使用应用程序，直到数据恢复过程完成。

尤其考虑到大多数最新产生的数据直到安全备份结束之前都存在丢失的风险，这就意味着备份（而不是存档）应该至少每天进行一次，或者每小时，甚至更短时间内进行一次。备份应该同时使用完整备份、增量备份，或者甚至是流式持续备份手段。

因此，当选择备份策略时，一定要考虑针对某个问题需要的恢复时间，以及可以丢失多少最新数据。

云计算环境下的需求

云计算环境引入了一系列独特的技术难点：

- 如果该环境使用了混合交易型和非交易型的备份和恢复方案，那么最终恢复的数据不一定是正确的。

- 如果某个服务必须在不停机的情况下更新，那么不同版本的逻辑可能同时并行操作数据。

- 如果所有其他有交互关系的服务不是同步更新的，那么在更新过程中各服务的不同版本之间可能会有多种组合，那么就更加增大了数据意外丢失和损坏发生的概率。

更多的是，为了保证扩展性，每个服务提供商都需要提供一定数量的API。这些API必须简单易用，否则就没有人会使用它们。但是同时这些API又必须可靠，以及支持以下特性：

- 数据本地性和缓存。
- 本地和全局的数据分布。
- 强一致性与/或最终一致性。
- 数据持久性、备份与灾难恢复。

否则的话，复杂应用将没有办法迁移到云上，而简单的应用程序，随着时间推移变得更为复杂，将会由于需要使用不同的、更复杂的API而进行完全重写。

前述的API在某种组合使用场景下也会造成某些问题。如果服务提供商没有预先解决这些问题，那么遇到这些问题的应用程序就必须要自己识别和解决这些问题。

保障数据完整性和可用性：Google SRE的目标

“保障持久性数据的完整性”是SRE的一个目标。这是一个很好的大方向，但是我们还需要一些更具体的，易于衡量的目标。SRE会定义一些关键指标，制定对应的测试，用来衡量我们运维的系统和流程的能力边界。同时在真实事故发生时，跟踪这些系统和流程的表现。

数据完整性是手段，数据可用性是目标

数据完整性指的是在其生命周期中，数据的准确性和一致性。从数据被记录的那一刻开始，一直到数据被访问的时候，数据应该保持正确，不会以某种未预知的方式改变。但是，这样就够了吗？

以一个E-mail服务提供商经历的长达一周的数据故障为例（参见文献[Kinc09]）。在长达10天的时间内，用户无法访问数据，这就意味着他们需要寻找到某种其他的、临时的解决方案来继续进行业务，同时期待他们可以很快取回他们现有的邮件地址、联系资料以及累积的历史信息。

然后，最糟糕的消息来临了：服务提供商宣布之前的预期有误，过去的历史信息和联系资料确实丢失了—消失了，无法再恢复。一系列保障数据完整性的系统错误导致了所有备份资料都无法用于恢复。愤怒的用户只有两个选择，要么长期使用之前的临时解决方案，要么彻底抛弃该服务商。

但是，故事还没有完！在宣布数据彻底丢失的数天之后，提供商又宣布了用户的个人信息最后可以恢复。最后结果是数据没有完全丢失，这次事故仅仅只是一次事故而已，这下没问题了吧！

但是，当然不是这样。用户数据虽然恢复了，但是由于太长时间无法访问，已经失去了很多意义。

这个例子的中心思想是：从用户的角度来看，仅仅保障数据完整性，而没有保障数据的可用性是没有意义的。

交付一个恢复系统，而非备份系统

针对系统进行备份是一个长期被忽视、被拖延的系统管理任务。任何人都不会将备份作为一个高优先级任务进行—备份需要长期消耗时间和资源，却不能带来任何现在可见的好处。由于这个原因，在备份策略具体实施中如果出现了某些被忽视的细节问题，大家一般都能理解。甚至有的人说，就像其他针对低可能性的危险的保护措施那样，这样的态度是必然发生的。这里的核心问题其实是，我们没有意识到备份措施防护的问题虽然是不太可能发生的，但却是会造成严重问题的。当该服务的数据不可用时，备份系统对整个服务、整个产品，甚至整个公司来说，都是生死攸关的。

相对于关注没人愿意做的备份任务来说，我们应该通过关注更重要

的（但并不是更简单的）恢复任务来鼓励用户进行备份。备份其实就像纳税一样，是一个服务需要持久付出的代价，来保障其数据的可用性。我们不应该强调应该纳多少税，而是应该强调这些税会用来提供什么服务：数据可用性的保障。因此，我们不会强迫团队进行“备份”，而是要求：

- 各团队需要为不同的失败场景定义一系列数据可用性SLO。
- 各团队需要定期进行演练，以确保他们有能力满足这些SLO。

造成数据丢失的事故类型

如图26-1所示，从很高的角度来看，可以将事故分类为3个因子的24种组合。我们应该针对每种潜在的事故类型来设计数据完整性机制。这3个因子分别是：

根源问题

某种无法恢复的用户数据丢失是由以下几个因素造成的：用户行为、管理员的错误、应用程序的Bug、基础设施中的问题、硬件故障和部署区的大型事故。

影响范围

有些数据丢失是大规模的，同时影响很多实体。有些数据丢失是非常有针对性的，仅仅是一小部分用户的数据损坏或者丢失。

发生速度

有些数据丢失是一瞬间造成的（例如，100万条数据在一次操作中被替换成了10条）。而有些数据丢失是缓慢持续进行的（例如，每分钟丢失10条数据，但是持续了一周时间）。

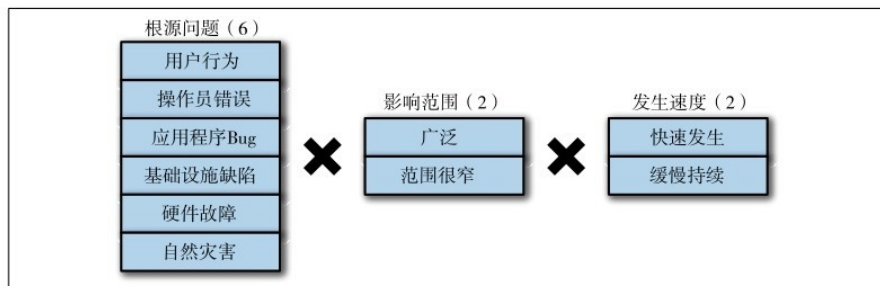


图26-1：数据完整性事故故障分类图

有效的故障恢复计划必须覆盖所有这些因子的全部有效组合。某个针对缓慢进行的应用Bug的恢复计划可能无法在全部数据丢失的情况下发挥作用。

Google根据一次针对19次数据恢复过程的研究得出，最常见的用户可见数据丢失场景是由于数据删除和软件Bug造成的引用完整性问题。其中遇到的最困难场景是，由于包含Bug的版本在更新数周、甚至数月之后才发现在生产环境中造成的数据损坏或者丢失。因此，Google所采用的保障机制也都是相应地针对这些类型的丢失设计的。

从这种事故中恢复，一个大型应用程序经常需要为数以百万计的用户恢复数天、数周，甚至数月的数据。这个应用程序甚至需要将每块数据恢复到一个特定的时间点。这种数据恢复场景在Google之外被称为“时间点恢复（point-in-time）”，在Google内部被称为“时间旅行”。

能够提供这种时间点恢复能力，同时能够覆盖该应用所使用的ACID与BASE类型的数据存储，同时还能够满足严格的在线时间、延迟、扩展性、更新速率，以及成本的解决方案就像神话中的怪兽一样，几乎不可能存在。

完全用自己的研发力量来试图解决这个问题意味着在创新速度上受损。很多项目最后都采用了一种分级的备份策略，但是不提供这种时间点恢复的能力。例如，应用程序所采用的底层API可能可以支持多种数据恢复手段。昂贵的本地“快照”可以针对软件Bug提供有限的保护，因为它们可以用来进行快速恢复。所以我们可以几个小时进行一次“快照”，然后将它们保留数天时间。而完整备份和增量备份可能每两天进行

一次，可以保存更长时间。如果这些工具中的一个或多个能够做到基于时间点的恢复，那就更好了。

在使用云计算API之前，一定要先考虑该API可选的数据恢复能力。有时候，这些API可能无法做到时间点恢复，但是也必须能够做到分级备份能力。如果两种特性都可用，那么就应该同时使用两种特性。

维护数据完整性的深度和广度的困难之处

在设计数据完整性保障机制时，必须要认识到复制机制和冗余并不意味着可恢复性。

扩展性问题：全量、增量以及相互竞争的备份和恢复机制

当别人问你“是否有备份”的时候，一个经典的错误回答是“我们有比备份更好的机制—复制机制！”复制机制很重要，包括提高数据的本地性（locality），保护某个部署点单点事故等。但是有很多数据丢失场景是复制机制无法保护的。一个自动同步多个副本的数据存储多半会在你发现问题之前，将损坏的数据记录以及错误的删除动作更新到多个副本上。

为了应对这个问题，我们可能会将数据定期导出成其他某种格式，例如将数据库导出成一个本地文件。这种机制可以在一定程度上针对用户错误和应用程序层面的Bug提供保护，但是仍然无法保护更底层的问题。同时，这种机制引入了几种新的危险—数据转换过程中的Bug（双向的），本地文件的存储问题，以及两种格式之间潜在的语义区别。假设某种0-day攻击[91]在系统底层—如文件系统或者设备驱动—中出现。任何依赖于该组件的副本，例如和数据库文件处于同一个文件系统的导出备份文件也会受到相同的影响。

因此，我们可以看到，多样性是关键：针对层X的故障恢复，需要在X层保存多样化的恢复副本。针对某种媒介的隔离可以预防媒介导致的问题：在磁盘设备驱动中的某个Bug，不太可能会影响到磁带备份。如果可能的话，我们应该把关键数据保存在黏土板上。[92]

对数据新鲜程度和复原完整程度的要求会与保护的周全性相互竞

争。在技术栈越底层的地方进行数据快照所需要的时间越长，也就意味着复制的频率越低。在数据库层，某个交易可能只需要几秒时间复制，然而将数据库快照导出成一个文件可能需要40分钟的时间，一个完整的文件系统备份可能需要几个小时。

在这个场景中，当还原最后一个快照文件时，我们可能丢失最近40分钟的数据。而一个文件系统的镜像还原可能造成几个小时的信息丢失。而且一般来说，还原与备份所需时间相同，所以加载这个数据可能还需要花费数个小时。能够最快地还原最新鲜的数据当然是最理想的情况，不幸的是，在某些故障场景下这不一定能够做到。

保留期

保留期—也就是数据备份保存的时间—是另外一个需要考虑的因素。

虽然用户或者管理员可能很快会发现某个数据库变空了，但是某些缓慢发生的数据丢失可能需要一段时间才能被注意到。在后面这个场景中恢复数据就需要向前追溯一段时间的数据。当恢复这种数据时，将恢复数据与现有数据进行合并会将整个恢复过程变得更复杂。

Google SRE保障数据完整性的手段

就像我们假设Google的底层系统经常出问题那样，SRE同样假设任何一个数据保护机制都可能在最不适合的时间出现问题。在所依赖的软件系统不停改变的情况下保障大规模数据的完整性，需要很多特定选择的、相互独立的手段来各自提供高度保障。

24种数据完整性的事故组合

由于数据丢失类型很多（如上文所述），没有任何一种银弹可以同时保护所有事故类型，我们需要分级进行。分级防护会引入多个层级，随着层级增加，所保护的数据丢失场景也更为罕见。图26-2显示了某个对象从软删除到彻底摧毁的过程，以及对应的分级数据恢复策略。



图26-2：某个对象从软删除到彻底摧毁的过程

第一层是软删除（soft deletion）（或者是某些API提供的“懒删除”机制）。这种类型的保护在实践中被证实是针对意外数据删除的有效手段。第二层，是备份和对应的恢复机制。第三层，也就是最后一层，更多信息参见本章后面“第三层：早期预警”小节。在这三层中，如果同时还有复制机制，那么在某些场景下对数据恢复是很有用的（但是数据恢复计划不应该依赖于复制机制）。

第一层：软删除

当更新速度很快，同时隐私很重要的时候，大规模数据丢失和损坏通常是由应用程序的Bug造成的。事实上，数据删除逻辑的Bug非常常见，以至于快速复原一定时间内的删除操作是针对永久性数据丢失的第一道防线。

任何一个保障用户隐私性的产品都必须允许用户删除全部或者一部分数据。这种产品不可避免地要处理误删除场景。允许用户恢复他们的数据（例如，使用回收站机制）可以降低出现频率，但是不能彻底消除这种场景，尤其是当该服务允许第三方插件删除数据的时候。

软删除机制可以大幅度减少支持人员的压力。软删除意味着被删除的数据立刻被标记为已删除，这样除了管理后台代码之外其他代码不可见。管理后台这里可能包括了司法取证场景、账号恢复、企业管理后台、用户支持，以及在线排错等。在用户清除自己的应用中的垃圾箱时应该采用软删除，同时提供某种支持工具允许有授权的管理员恢复误删除的文件。Google在使用量最高的工具上都实现了这种机制，否则用户支持所带来的压力是无法持续的。

我们可以将这种软删除策略继续延伸，直接提供给用户数据恢复机制。例如，Gmail的垃圾箱允许用户恢复30天之内删除的信息。

另外一个常见的数据误删除场景是由于账号被劫持造成的。在这个场景中，劫持用户账号的人常常会先将用户原始数据删除，再用这个账号来发送垃圾或者进行其他非法活动。当我们将用户误删除和账号劫持两个场景结合起来，在应用程序层内，或者其下某层实现软删除机制的需求就非常明显了。

软删除还意味着一旦数据被标记为已删除，在某段时间以后会真的被删除。这个时间的长度，尤其是在有很多短期数据的情况下，取决于某个组织的政策和相关的法律条文、可用的存储空间和存储成本，以及产品的价格和市场定位等。常见的软删除时间是15、30、45或者60天。在Google的经验中，大部分账号劫持和数据完整性问题都会在60天内汇报或者检测到。因此，为软删除保存超过60天的需求可能没有那么强烈。

Google同时发现最严重的数据删除案例经常是由于某个不熟悉现有代码的开发者实现新的删除逻辑时造成的。尤其是在开发某种离线批处理数据流水线的时候（例如，某个离线的MapReduce，或者Hadoop流水线）。于是，在设计接口的时候，最好能够使得不熟悉现有代码的开发者无法（或者非常困难）绕过软删除逻辑。一个有效方法是让云计算API提供内置的软删除和恢复删除API，同时确保启用这项功能。[93]再好的盔甲也要穿上才能起作用。

面向直接用户的Gmail或者Google Drive的删除功能都采用了软删除策略，但是云计算API类型的服务怎么办呢？假设该项服务已经提供了可编程的软删除和还原机制，同时有比较合理的默认值，剩下的误删除场景就是由内部开发者或者是外部开发者错误操作造成的数据丢失了。

针对这种场景，有时候可能会再增加一层软删除机制，我们称之为“懒删除”机制。可以将“懒删除”认为是某种幕后清理机制，由存储系统控制（与此相比，软删除是由程序或者服务直接控制的）。在懒删除场景下，由某个云计算应用程序删除的数据马上对应用程序不可用，但是由云计算服务提供商保留几周时间再彻底销毁。懒删除不一定在所有

的策略中都适用：在一个短期数据很多的系统中保存长期懒删除数据可能成本会很高，甚至在某些需要保障已删除数据的销毁时间的系统中不可行（例如某些隐私敏感的程序）。

最后，针对第一层防线的总结：

- 应用中实现一个回收站机制，作为用户错误的主要防护手段。

- 软删除机制是针对开发者错误的主要防范手段，以及用户错误的次要防范手段。

- 在面向开发者的服务中，懒删除机制是针对内部开发者错误的主要防范手段，是针对外部开发者错误的次要防范手段。

历史记录功能属于哪种？有些产品允许将某个对象恢复到之前某个状态。当用户直接可用这种功能时，这其实就是回收站机制的一个变种。当这种功能针对开发者可用时，取决于具体实现，可以用来替代软删除机制。

对Google来说，历史记录功能在恢复某些特定数据损坏场景下比较有用，但是在恢复大部分数据丢失的场景中（包括误删除、人工以及程序化都算）都没有作用。因为在某些历史记录功能的实现中，将删除作为一个特例处理，要求之前的历史状态也要被删除，而不是仅仅修改历史记录中的一条记录。为了针对这些场景提供合理的防护，我们也需要将懒删除与/或软删除机制应用于历史记录上。[94]

第二层：备份和相关的恢复方法

备份和数据恢复是软删除之后的第二道防线。这一层中最重要的理念是：备份不重要；最重要的是恢复。对“恢复”提供支持，应该是主导备份系统设计的关键。

由此可见，在设计备份还原系统时，必须考虑：

- 使用哪种备份和还原方法。

- 通过全量或者增量备份建立恢复点（restore point）的频率。

- 备份的存储位置。
- 备份的保留时间。

另外一个关键问题是：在一次数据恢复中能够损失多少最近数据。能够损失的数据越少，增量备份策略就越重要。在Google的一个极端的例子中，旧版的Gmail采用了一个准实时的流式备份系统。

就算不考虑成本问题，频繁地进行全量备份也是非常昂贵的。最主要的是，这会给线上服务用户的数据存储造成很大的计算压力，以至于影响服务的扩展性和性能。为了缓解这种压力，我们需要在非峰值时间段进行全量备份，同时在繁忙时间段进行增量备份。

在此之外，故障恢复的时间很重要。所需的故障恢复时间越短，备份存放的位置越需要本地化。Google通常只会将那些恢复很快，但是成本很高的“快照”存储很短的一段时间（在同一个存储实例中）。[95]同时，将其他长时间的备份存放在处于同一个（或者很近的）数据中心的分布式存储上。这样一个策略本身并不能保护单个部署点的故障，所以这些备份经常会在一段时间后被转移到其他离线存储上，从而给更新的备份腾出地方。

备份应该保存多久呢？备份策略随着保存时间的增长，成本会上升，但是同时可恢复的时间也会提高（注意，这里也会受边际效应的影响）。

Google的经验证实，应用程序Bug造成的低等数据损坏，或者应用程序内部的删除Bug需要的恢复时间最长，因为这些Bug可能在引入之后几个月才被发现。这样的场景意味着我们需要能够恢复得越久越好。

但是，在高速更新的开发环境中，代码和数据格式的改变可能使得旧的备份很快就无法使用了。更重要的是，试图恢复不同部分的数据到不同的时间点可能是不可能的，因为这需要同时操作多个备份。但是，这恰恰是这种低等数据损坏和丢失场景中最需要的。

本章后面“第三层：早期预警”一节中描述的策略都是为了加速检测应用程序Bug造成的数据问题，以便于减少需要这种复杂恢复场景的需

求。然而，我们在无法预知问题类型之前，如何确定合理的备份保存周期呢？Google将大部分服务的备份周期定于30～90天之间。每个服务必须通过针对数据丢失的容忍度设计和早期预警系统的研发投入来保障落入这个区间。

将以上针对24种组合的预防建议总结得出：以合理成本满足一系列广泛的数据恢复场景都需要分级备份。第一级备份是那些备份频率很高，并且可以快速恢复的备份。这些备份保存在与线上数据存储距离最近的地方，使用与数据存储相同或者是非常相似的存储技术。这样可以为大部分的软件Bug，以及开发者错误的场景提供保护。由于成本相对较高，备份在这一层只会保留数小时，最多10天之内，恢复时间在几分钟内。

第二级备份的频率较低，只保留一位数或者两位数字的天数。第二级备份保存在当前部署点的随机读写分布式文件系统上。这些备份可能需要数个小时来备份，是用来为服务所使用的数据存储出现的相关问题提供额外保护的，但是要注意，这无法为用来保存备份的技术栈自身的问题提供保护。这一级别的备份也可以用来针对那些检测时间太晚的应用程序Bug，以至于无法使用上一级别的备份恢复的问题。如果新版本的发布时间一般是一周两次，那么保存一周到两周再删除这种备份可能是合理的。

接下来的级别会使用冷存储，包括固定的磁带库以及异地备份存储设施（不管是磁带，还是磁盘）。这些级别的备份可以保护单个部署点级别的故障，例如数据中心电源故障，或者是Bug造成的分布式文件系统的数据损坏。

在不同的层级中转移大量数据是很昂贵的，但是后续层级的数据存储容量并不会与生产系统线上数据容量竞争。所以，后续层级的备份数据通常产生的频率很低，但是可以保存的时间更长。

额外一层：复制机制

在理想世界中，每个存储实例，包括那些保存了备份的存储系统都应该具有复制机制。否则，就有可能在数据恢复过程中，才发现备份文

件自身丢失了数据，或者发现保存备份文件的那个数据中心正在进行维护。

随着数据量的增长，不是每个存储系统都可以进行复制。在这种情况下，将不同层级的备份存放在不同部署点上是更合理的，这样每个部署点不会同时出现故障。同时，应该将备份文件放置在某种冗余机制之上，例如RAID、Reed-Solomon纠错码，或者GFS类型的复制机制之上。[96]

当选择冗余系统时，不要选择一种不常用的系统。因为这种系统的可靠性只能通过本身就很罕见的数据恢复过程来测试。正确的做法是，选择一个非常常见，并且被许多用户持续使用的系统。

1T vs.1E ：存储更多数据没那么简单

针对T级别（Terabytes）的流程和手段在E级别（Exabytes）并不能工作得很好。在几个GB的结构化数据上进行校验、复制和进行端到端测试可能有些挑战。但是只要我们有足够的数据格式信息，了解数据的交易模式等，这个过程并不会有特别多的难点。一般来说，我们只需要找到足够的机器资源，遍历全部数据，执行某种校验逻辑就可以了，可能还需要有足够的存储空间放置数据的几个副本。

现在，我们用同样的策略来校验700 Petabytes的结构化数据。就算我们使用一个理想化的SATA 2.0接口（300MB/s的性能），仅仅是遍历一遍所有数据进行最基本的校验也将需要80年。而进行几次全量复制，就算我们有足够的存储媒介也至少需要同样长的时间。备份的恢复时间，再考虑到一些后续处理时间则会需要更长。这样看来，我们可能会需要将近一个世纪的时间来恢复这些数据，别忘了这些数据在恢复的时候已经是至少80年以前的了。显而易见，这个策略需要重新设计。

处理海量数据，最重要也最有效的方式是给数据建立一个“可信点”—也就是一部分数据校验过之后，由于时间等原因变成不可变数据了。一旦我们确信某个用户的资料或者交易记录已经是不会再变的了，就可以校验并且进行合适的复制，以便未来恢复。接下来，我们可以进行增量备份，其中仅仅包含自从上次备份后新增或者更改过的数据。这

种技术可以将备份时间缩减到与主要处理逻辑的吞吐量在一个数量级内。这样的话，经常性的增量备份就可以替代需要80年的巨型校验和复制任务了。

然而，上文说过，我们最关注的是恢复，而不是备份。假设三年前进行了一次全量备份，然后每天进行增量备份。完全恢复一份数据就需要顺序处理大概1000个相关性非常强的备份。每一个相关性很强的备份都会增加故障发生的风险，这还没有考虑到后勤和计算上的成本。

另外一个降低复制和校验任务的时间总量的方法是分布式计算。如果我们将数据合理分片，可以将N个任务并行进行，每个任务可以负责复制和校验1/N的数据。这样需要在设计部署阶段预先进行一些考量，以便于：

- 正确平衡数据分片。
- 保证每个分片之间的独立性。
- 避免相邻并行任务之间的资源抢占。

通过水平分割负载，同时按时间维度进一步限制垂直数据量，我们可以将80年的处理时间降低几个数量级，这样恢复策略才能有效工作。

第三层：早期预警

“脏”数据并不会一直静止不动，它们会在整个系统中传播。针对不存在的或者是已经损坏的数据的引用可能会被复制好几份。随着每次更新，数据存储中数据的整体质量会持续下降。关联性的交易和潜在的数据格式变化使得从某个指定备份中还原数据的能力随着时间大幅下降。越早检测到数据丢失，数据的恢复就越容易，也越完整。

云计算开发者面临的挑战

在高创新速度的环境中，云计算应用程序和基础设施服务面临着很多数据完整性的挑战，例如：

- 不同的数据存储之间的引用完整性。

- 数据结构的改变。
- 旧代码。
- 无停机时间的数据迁移。
- 与其他服务接口的不断变化。

如果不主动花费一定的精力在数据关系的管理上，任何一个不断增长的服务的数据质量肯定是会不断下降的。

一般来说，初级的云计算平台开发者会选择使用一个分布式一致性的存储API（例如Megastore），也就是将应用程序数据一致性的问题交给该API使用的分布式共识性算法来保障（例如，Paxos，见第23章）。开发者认为选择合适的API就足以保障应用的数据完整性了。于是，开发者通常将所有应用数据合并到单个确保分布式一致性的存储方案中，从而避免造成引用一致性的问题，哪怕这样会造成性能和扩展性的问题。

虽然分布式共识性算法理论上无懈可击，但是具体的实现经常充满了Hack、优化、Bug和一些猜测成分。例如：理论上来说，Paxos会忽略掉故障的计算节点，只要有满足法定仲裁数量的节点还存在就能够继续进行工作。但是，在实际实现中，具体的忽略逻辑要应对对应的超时、重试和其他灾难处理方法，这些都埋藏在具体的Paxos实现中（参见文献[Cha07]）。Paxos应该等待多长时间再忽略一个失去响应的节点？当某个具体的物理机器出现某个特定故障时（可能是暂时的），符合某种特定的时间，以及出现在某个具体的数据中心时，可能会有未定义的行为发生。应用程序的部署范围越大，那么受这种无法预知的不一致性的影响就越严重。如果这个逻辑对Paxos实现适用（Google的经验证明的确如此），那么对采用最终一致性的Bigtable等实现来说就更适用了（Google的经验再次证明）。不检查受影响的应用程序的话，没有任何办法可以100%　　确定数据是正确的：虽然我们相信存储系统的实现，但是还是要校验！

使这个问题变得更复杂的是，为了从低级数据损坏或者删除场景中

恢复数据，我们必须从不同恢复点的不同备份中恢复不同子集的数据。同时，我们还要考虑到在一个不断变化的环境中，代码和数据结构的变化对老的备份数据的影响。

带外数据校验

为了避免用户可见的数据质量下降，以及在无法恢复之前检测到低级的数据损坏以及数据丢失，我们需要一整套带外（out-of-band）检查和修复系统来处理数据存储内部和相互之间的数据问题。

大部分情况下，数据校验流水线被实现成一系列MapReduce任务或者是Hadoop任务。更常见的是，这些任务通常是在一个服务成功之后后续添加的。有些时候，这些流水线任务在服务达到某个扩展性节点的时候第一次被引入，随着架构的调整完全重写。Google针对上述每一种情况都构建了对应的校验器。

安排一些开发者来开发一套数据校验流水线可能会在短期内降低业务功能开发的速度。然而，在数据校验方面投入的工程资源可以在更长时间内保障其他业务开发可以进行得更快。因为工程师们可以放心数据损坏的Bug没那么容易流入生产环境。和在项目早期引入单元测试效果类似，数据校验流水线可以在整个软件开发过程中起到加速作用。

举一个具体的例子：Gmail拥有一系列数据校验器，每一个校验器都曾经在生产环境中检测到真实的数据完整性问题。Gmail开发者由于知道每个新引入的数据一致性问题都可以在24小时之内被检测到而感到非常安心。这些数据校验器的存在，结合单元测试及回归测试的文化使得Gmail开发者有足够的勇气每周多次修改Gmail生产存储系统的实现。

带外数据校验比较难以正确实现。当校验规则太严格的时候，一个简单的合理的修改就会触发校验逻辑而失败。这样一来，工程师就会抛弃数据校验逻辑。如果规则不够严格，那么就可能漏过一些用户可见的数据问题。为了在两者之间取得恰当的平衡，我们应该仅仅校验那些对用户来说具有毁灭性的数据问题。

例如，Google Drive的周期性校验文件内容与Drive文件夹中的列表

一致。如果不一致，那么某些文件可能缺少数据—也就是毁灭性的问题。负责开发Drive所用基础设施的工程师非常注重数据一致性，他们甚至将数据校验器增强为可以自动修复这种不一致情况。这个安全机制使得2013年发生的一次“天啊，所有文件都在消失”的意外情况变成了“我们可以放心回家睡觉，周一再来修复根源问题”的情况。这样，数据校验器提高了工程师的士气，避免了加班，也提升了功能上线的可预知性。

大规模部署带外检测器可能成本较高。Gmail计算资源的很大一部分都被用来支持每日数据检测的运行。使这个问题变得更严重的是，校验器本身可能会造成软件服务器缓存命中率的下降，这就会造成用户可见响应速度的下降。为了避免这种问题，Gmail提供了一系列针对校验器的限速机制，同时定期重构这些校验器，以降低磁盘压力。在某一次重构工作中，我们降低了60% 磁盘磁头的使用率，同时没有显著降低校验器覆盖的范围。虽然大部分Gmail检测器每天运行一次，但是压力最大的校验器被分为10 ~ 14个分片，每天只会运行一个分片。

Google云存储（compute storage）是另外一个数据量显著影响数据校验的例子。当带外校验器已经不能一天之内完成时，云存储工程师开发了一种更有效校验元数据的方法，而不再使用暴力方法校验。就像在数据恢复机制中那样，带外数据校验也可以分级进行。随着一个系统的规模不断扩展，每天能够运行的校验器越来越少。我们可以保证每天运行的校验器持续寻找毁灭性的问题，而其他更严格的校验器可以以更低频率运行，以便满足延迟和成本的要求。

分析校验器为何失败也需要很多精力。造成某项间断性不一致情况的原因可能在几分钟、几小时或者几天之内消失。因此，快速定位校验日志中的问题是非常关键的。成熟的Google服务给on-call工程师提供了非常完备的文档和工具，用来定位问题。例如，Gmail提供给on-call工程师的有：

- 一系列Playbook文章讲述如何应对某种校验失败的报警。
- 类似BigQuery的查询工具。
- 数据校验监控台。

一个有效的带外数据校验系统需要下列元素：

- 校验任务的管理。
- 监控、报警和监控台页面。
- 限速功能。
- 调试排错工具。
- 生产应急应对手册。
- 校验器容易使用的数据校验API。

大部分高速进行的小型开发团队都负担不起设计、构建和维护这样的系统所需的资源。就算强迫他们进行，最后的结果也经常是不可靠、不全面的，以及浪费性、一次性的方案，很快就会被抛弃。因此，最好的办法是单独组织一个中央基础设施组为多个产品和工作组提供一套数据校验框架。该基础设施组负责维护带外数据校验框架，而产品开发组负责维护对应的业务逻辑，以便和他们不断发展的产品保持一致。

确保数据恢复策略可以正常工作

一个电灯泡什么时候会坏掉？是在拨动开关而灯却不亮的那一瞬间坏的吗？当然不是—电灯泡可能早已经坏了，拨动开关时只是注意到了这个已经损坏的情况。而这时，你要面临一个漆黑的房间，可能还会扎到自己的脚趾（美国俚语，指自己给自己制造了障碍）。

同样的，成功进行一次数据恢复所需要的资料（通常是你的备份数据）可能也处于一个损坏的状态，在试图进行恢复之前我们并不知道。

如果能够在真正需要这些数据之前检测到数据损坏，我们就可以在灾难发生之前避免它们：我们可以再进行一次备份，准备更多的资源，或者修改SLO。那么，主动检测这些问题需要以下步骤：

- 持续针对数据恢复流程进行测试，把该测试作为正常运维操作的一部分。

- 当数据恢复流程无法完成时，自动发送警报。

数据恢复流程中任何一个环节都有可能出问题，只有建立一个完善的端到端测试才能让人晚上放心睡觉。即使最近刚刚成功进行过一次数据恢复，下次执行时某些步骤仍然可能出问题。如果读者从本章仅仅学到一个知识点，那就是一定要记住：只有真正进行恢复操作之后，才能确定到底能否恢复最新数据。

如果数据恢复测试是一个手工的、分阶段进行的操作，那么就不可避免地成为一个讨人嫌的麻烦事。这样会导致测试过程要么不会被认真执行，要么执行得不够频繁以至于作用有限。因此，在任何情况下，都应该追求完全自动化这些测试步骤，并且保证它们能持续运行。

数据恢复计划中需要覆盖的点是很多的：

- 备份数据是否是完整的、正确的—还是空的？
- 我们是否有足够的物理机资源来完整地完成整个恢复过程：包括运行恢复任务本身，真正恢复数据，以及数据后处理任务？
- 整个数据恢复过程能否在合理的时间内完成？
- 是否在数据恢复过程中监控状态信息？
- 恢复过程是否依赖于某些无法控制的元素—例如某个不是24×7随时可以使用的异地存储媒介？

Google不光发现过上述列表中提到的每一个问题，还意外发现了很多其他没有提到的失败场景。如果我们没有在日常测试中主动寻找这些问题，而是在每一次进行数据恢复的时候才意外发现，那么Google很多非常受欢迎的产品可能就不会存在了。

失败是不可避免的，不去主动寻找这些数据恢复失败的场景等于玩火自焚。只有通过进行测试来主动寻找这些弱点，才能够提前修复，关键时刻才不至于追悔莫及！

案例分析

正如我们预测的那样，真实生活给我们提供了无数个既不幸又不可避免的机会来检验数据恢复系统，以及对应的流程的可靠程度。接下来我们会讨论两个非常有名，并且非常有趣的场景。

Gmail—2011年2月：从GTape上恢复数据（磁带）

作为我们学习的第一个数据恢复案例，它在很多方面都很独特：首先是数量众多的组件同时失败造成的数据丢失，其次，这次数据恢复过程是Google数据安全最后一道防线—离线磁带备份系统历史上使用量最大的一次。

2011年2月27日，星期日，深夜

Gmail备份系统触发了一条报警信息，里面带有一个电话会议的号码。我们长时间以来最害怕的事情终于发生了—这其实也正是备份团队组建的原因—Gmail系统丢失了用户数据，大量的用户数据。即使系统中存在很多安全防护措施，很多内部检查机制，以及大量的冗余，数据还是从Gmail系统中消失了。

这是GTape，为Gmail量身定制的一个全球磁带备份系统历史上最大的一次应用—恢复线上用户数据。还好，这种数据恢复以前曾经进行过测试，而且本次丢失的原因与之前的测试也相符。因此，我们可以：

- 很快计算出恢复绝大部分受影响的用户数据的预计时间。
- 在该预计时间之后几个小时内恢复其他剩余全体用户。
- 在全部过程预计完成时间之前应该可以恢复超过99%的数据。

这里能够建模并且产生一个预期时间并不是靠运气，而是长时间持续进行规划的结果，是坚持实行最佳实践，努力工作，以及协作才能做到的。看着每一个涉及的组件都成功完成任务，是一件很有成就感的事。Google 通过严格按照前文 “深度多层防御”和“应急事件处理流程”中提到的最佳实践设计恢复计划，最终做到及时地恢复了用户的数据。

当Google最终宣布我们是通过之前未宣传过的磁带备份系统恢复的

数据时（参见文献[Sol11]），公众反响非常大。大家认为既然Google拥有大量的磁盘和快速网络，那么一定可以多存几个副本吧？当然，Google确实有这些资源，但是“深度多层防御”理念中要求提供多层防御，以便在任何一个单独的防护措施失效时仍能工作。针对Gmail这种在线系统的备份要从两个层面考虑：

- 针对Gmail内部冗余及备份子系统的故障。
- 一个会影响底层存储媒介（磁盘）的0-day设备驱动或者文件系统中的问题。

本次故障正是第一个场景的再现—Gmail虽然有内部工具恢复丢失的数据，但是由于本次丢失数据太多，该工具已经不能正常运行了。

Gmail数据恢复过程中受到最多表扬的部分是大量的团队协作以及平滑的沟通过程。很多完全和Gmail无关的团队都提供了帮助。这次数据恢复过程通过建立一个完整的中央计划妥善协调了在全球范围内分布式进行的费力工作—如果没有平时经常性的演习是不可能做到的。Google强调这种灾难发生的必然性—甚至期盼它们尽快发生。因此，我们不仅针对可预知的灾难建立起处理计划，同时也引入了一些随机无差异的失败场景。

简单来说，虽然我们一直都知道这些最佳实践是很重要的，本次故障则实际证明了这一点。

Google Music—2012年3月：一次意外删除事故的检测过程

这里讨论的第二个故障案例的特殊点在于海量数据存储所带来的后勤方面的挑战：在哪里存放5000盘磁带，以及如何能够迅速地（甚至是可行的）从离线媒介中读出数据—而这一切还要发生在一个合理的时间范围内。

2012年3月6日，星期二，下午

发现问题

一个Google Music用户汇报某些之前播放正常的歌曲现在无法播放

了。Google Music的用户支持团队通知了工程师团队，这个问题被归类为流媒体播放问题进行调查。

3月7日，负责调查此事的工程师发现无法播放的歌曲的元数据中缺少了一个针对具体音频数据文件的指针，于是他就修复了这个歌曲的问题。但是，Google工程师经常喜欢深究问题，也引以为豪，于是他就继续在系统中查找可能存在的问题。

当发现数据完整性损坏的真正原因时，他却差点吓出心脏病：这段数据是被某个保护隐私目的的数据删除流水线所删掉的。Google Music的这个子系统的设计目标之一就是在尽可能短的时间内删除海量音频数据。

评估问题严重性

Google的隐私策略强调保护用户的个人数据。在Google Music服务中，该隐私策略要求音乐文件，以及对应的元数据需要在用户删除它们之后在合理时间范围内在系统中彻底删除。随着Google Music使用量呈指数级增加，数据量也飞快地增长，以至于原始的删除机制被抛弃了，在2012年进行了重新设计。在2月6日那一天，更新过的数据删除流水线任务进行了一次运行，当时看起来没有任何问题，于是工程师批准了流水线任务的第二阶段执行—真正删除对应的音频数据。

然而，真的是这次删除任务造成的问题吗？该工程师立刻发出了一个最高级别的警报，同时通知了相关的工程师经理，以及SRE团队。Google Music的开发者和SRE组建了一个小团队专门调查这个情况，同时该数据删除流水线也被暂时停止了，以防问题范围扩大。

接下来，手动检查分散在多个数据中心内部的百万条，甚至数十亿条文件元数据信息显然是不可能的。该团队匆忙编写了一个MapReduce任务来评估问题的影响范围，焦急地等待着任务执行结果。3月8日，任务终于完成了，看到结果时所有人都惊呆了：该流水线任务大概误删了60万条音频文件，大概影响了2.1万用户。由于这个匆忙编写的检测程序还简化了一些检测步骤，实际的问题只会更严重。

距离有问题数据删除流水线第一次运行已经超过一个月了，正是那次运行删除了几十万条不该删除的音频数据。现在还有恢复数据的希望吗？如果这些数据无法恢复，或者不能及时恢复，用户还会使用Google Music吗？我们怎么能没有注意到这个问题的发生呢！

问题的解决

定位**Bug**和数据恢复的并行进行。解决问题的第一步是定位真正的Bug根源，以便了解Bug产生的原因和过程。如果不修复根源问题，任何数据恢复工作都是做无用功。团队承受着来自用户的压力要求我们重新启用数据删除流水线，而无辜用户将会继续丢失刚刚购买的音乐，甚至是他们自己辛苦录制的音乐。这个22条军规[97]的场景的唯一出路，就是修复根源问题，并且要快。

然而数据恢复流程也没有任何时间可以浪费。音频文件虽然备份到磁带上，但是和Gmail不同，Google Music由于数据量太大，加密过的备份磁带是由卡车运往异地存放于独立的存储设施中的——这些设施可以存放更多磁带。为了能够更快地为受影响的用户恢复数据，团队决定在调查根源问题的同时并行进行异地备份磁带的恢复操作（这将耗时很长）。

工程师分为两组。经验最丰富的SRE负责数据恢复，同时开发者负责分析数据删除逻辑代码，试着修复根源问题。由于问题的根源尚不清晰，整个数据恢复过程会按数个阶段进行。首先，第一批超过50万条音频数据被选中进行恢复，负责磁带备份系统的团队在美国西海岸时间3月8日下午4:34收到了恢复通知。

整个事件中有一个好消息：公司组织的年度灾难恢复演习刚刚在几周前结束（参见文献[Kri12]）。磁带备份团队由于在DiRT演习中刚刚评估过子系统的能力和限制，刚刚制作了对应的数据恢复新工具。利用这个新工具，新团队开始缓慢地首先将几十万条音频文件和磁带备份系统中注册的备份文件一一对应，然后再将备份文件与具体的物理磁带一一对应。

通过这个方法，团队评估得出第一批恢复过程需要将5000盘磁带用

卡车全部运回来。并且之后数据中心的技术人员需要清理出足够的场地来放置这些磁带。接下来还要通过一个极为复杂和耗时的流程来从磁带上读取这些数据。同时，还要处理磁带损坏、磁带读取设备损坏，以及其他不可预料的问题。

更不幸的是，60万条丢失的数据中只有436,223条数据存在备份，也就意味着其他161,000条数据在备份之前就已经丢失了。数据恢复团队最后决定先启动恢复流程，再想办法恢复这161,000条无备份数据。

同时，负责调查根源问题的团队找到了一个潜在问题，结果却最终被证伪：他们本以为Google Music的底层的数据存储服务提供了错误数据，以至于数据删除流水线删除了错误的数据。但是经过详细调查，这个可能性被排除了。这个团队只能继续寻找这个看起来好像不存在的Bug。

第一批数据的恢复。数据恢复团队确认了备份磁带之后，第一批数据恢复在3月8日正式开始了。从分布在异地离线存储设施中的几千盘磁带中请求超过1.5PB的数据看起来很难，从这些磁带中真正读取数据就更难了。Google自己定制的磁带备份软件设计时没有考虑过这么大的恢复任务，所以第一批数据的恢复过程被切分成5,475个小任务。如果由人每分钟输入一条命令，光输入就需要超过三天的时间。人在这个过程中很可能还会引入很多错误。仅仅是这一点准备工作，就需要SRE写代码来完成。[98]

在3月9日午夜的时候，SRE终于将5,475个恢复请求输入了系统中。磁带备份系统计算了4个小时，得出5,337盘磁带需要从异地存储设施中恢复。8小时后，一系列卡车运载着这些磁带抵达了数据中心。

在卡车奔波在路上时，数据中心的技术人员清空了几个磁带管理库，撤下了几千盘磁带以便给这次大型数据恢复工作让路。接下来，工作人员又开始一点一点手动将刚刚抵达的几千盘磁带装进磁带管理机中。由过去的DiRT演习证明，这种手动装载的方式要比磁带管理机厂商提供的基于机器手臂模式的装载方式快很多。3个多小时后，磁带管理机终于恢复了运行，开始将几千个恢复任务逐渐写入分布式存储中。

虽然之前有DiRT的经验，1.5PB的海量数据恢复还是比预计时间长了2天。到3月10日早晨，436,223个文件中只有74%的文件从3,475盘磁带上成功转移到了数据中心中的分布式存储中。其他1,862个磁带在第一次运输中完全漏掉了。更糟的是，整个恢复过程还被17个坏磁带拖慢了。由于已经预计到会有磁带损坏，写入时已经采用了带冗余的编码。于是还需要额外安排卡车去取回这些冗余磁带，一并再去取回第一次漏掉的1,862盘磁带。

到3月11日早晨，超过99.95% 的恢复任务已经完成了，剩余文件的冗余磁带的取回也在进行中。虽然数据已经安全地存到了分布式存储中，但还有额外的数据恢复过程需要完成，才能使用户真正可以访问。Google Music团队开始在数据的一小部分上进行数据恢复过程的最后步骤，同时不停校验结果以确保这个恢复过程能够成功执行。

同时，Google Music生产系统的报警信息又来了，虽然与数据丢失不相关，但却是非常严重的用户直接可见的生产问题—该问题又消耗了恢复团队整整两天的时间。在3月13日，数据恢复过程得以继续，最终436,223条文件再一次可以被用户访问了。仅仅花了7天时间，我们恢复了1.5PB的用户数据，7天之内的5天用于真正的数据恢复操作。

第二批数据恢复。当第一批数据恢复过程结束之后，团队开始关注如何恢复剩下的161,000条音频数据，这些数据在备份进行之前就被错误地删除了。这些文件中的大部分都是商店出售的，或者是推广性质的音频，原始文件都还存在。这些文件很快被恢复了。

但是，161,000条中的一小部分是由用户上传的。Google Music团队发出了一条指令，远程遥控这些受影响的用户的Google Music客户端软件自动重新上传3月14日之后的文件。整个过程持续了一周时间，最终全部数据恢复工作结束。

解决根源问题

最终，Google Music团队找到了重构过的数据删除流水线中的Bug。为了更好地解释这个Bug，我们需要先了解大型离线数据处理系统的演变过程。

对一个包括很多子系统和存储服务的大型服务来说，彻底删除已经标记为删除的数据需要分多个阶段进行，每个阶段操作不同的数据存储服务。

为了使数据操作服务更快地结束，整个过程可以并行运行在几万台机器上，这会给很多子系统造成很大压力。这种分布式操作会影响到用户，同时导致某些服务由于压力过大而崩溃。

为了避免这些问题，云计算工程师通常会在第二阶段保存一些短期数据，用它们来进行数据存储。如果没有仔细调整过这些短期数据的生命周期，这个方法可能会引入数据竞争问题。

例如，该任务两个阶段被设计成严格隔离的，运行时间相隔3个小时。这样第二阶段的代码可以大幅度简化某些逻辑，否则这个阶段的处理逻辑可能很难并行化。但是随着数据量的增长，每个阶段需要更多时间才能完成。最终，当初设计时候的某些假设，在这个新条件下就不再成立了。

一开始，这种数据竞争问题可能只会影响到一小部分数据。但是随着数据数量的增加，越来越多的数据可能会受到该数据竞争的影响。这种场景是随机化的—对绝大部分数据，绝大部分时间来说，整个流水线任务是正确的。但是一旦发生数据竞争问题，错误的数据就会被删除掉。

Google Music的数据删除流水线任务设计时考虑了协调机制，并且加入了很多防错机制。但是当整个流水线的第一阶段花费的时间越来越长时，工程师加入了很多性能优化以保障整个Google Music的隐私策略不受影响。结果，这导致了这种数据竞争问题发生的概率的提升。当整个流水线任务重构时，这个概率被再次大幅提高了，直接导致了这个问题经常性地发生。

经历过这次数据恢复之后，Google Music团队重新设计了该流水线任务，彻底消除了这种数据竞争问题出现的可能性。同时，我们增强了生产系统的监控和报警系统，使得它们可以提前检测类似的大规模删除问题，以便在用户发现之前检测和修复这类问题。[99]

SRE的基本理念在数据完整性上的应用

保持初学者的心态

大规模部署的、复杂的服务中会产生很多无法完全被理解的Bug。永远不要认为自己对系统已经足够了解，也不要轻易将某些失败场景定性为不可能。我们可以通过“信任仍要验证”、“纵深防御”等手段来保护自己。（注意，这里可不是说让一个初学者来管理上文说的那个数据删除流水线！）

信任但要验证

我们依赖的任何API都不可能一直完美工作。不管测试有多么复杂，工程质量有多么高，API都一定会存在某些问题。应该通过使用带外数据检测器来检查数据最关键、最核心的部分，哪怕API语义中说明了这些问题不存在。要记住，完美的理论不代表实现也是完美的。

不要一厢情愿

不经常使用的系统组件一定会在你最需要的时候出现故障。数据恢复计划必须通过经常性的演习来保障可用性。由于人类天生不适合持续性、重复性地进行测试活动，自动化手段是必备的。然而，如果没有专人负责数据问题，而是让有其他工作的工程师兼职来弄，那么中间一定会出现问题。

纵深防御

就算是最安全的系统也会受到实现中的Bug，以及操作错误的影响。保障数据完整性问题能够及时恢复的重要条件是能够很快检测到问题的存在。在一个不断变化的环境中，每个策略都可能失败。最好的数据完整性保障手段一定是多层的—多个保障手段彼此覆盖，能够用合理的成本来覆盖非常广泛的失败场景。

定期重新检查和重新审核

数据今天是安全的，并不意味着明天还是安全的。你的服务，甚至基础设施都在不停地变动，我们必须要验证之前的假设和流程在新的环

境下仍然适用。例如下面这个例子：

前文提到的莎士比亚服务反响很好，用户群在稳定增长。在服务构建之初，没有真正关注过数据完整性问题。因为虽然我们不想返回错误结果，但是就算Bigtable上面的索引丢失，我们也可以很快地用一个MapReduce任务恢复。整个恢复时间很短，所以我们从来也没有备份过索引。

后来，团队新增了一个功能——允许用户增加文字标记。突然之间，我们的数据再也无法被迅速恢复了，而数据对用户来说也变得越来越重要。因此，我们需要重新审视我们的复制策略——不光是为了延时和带宽，现在数据安全性也很重要了。因此，我们需要创建和测试一套数据备份与恢复的流程。这套机制作为DiRT演习的一部分要持续测试，以保障我们可以在SLO规定的时间内将用户的信息从备份中恢复。

小结

数据可用性必须作为任何以数据为中心的系统的首要重点。与其关注于具体的实现细节，Google SRE采用了一种与软件测试类似的流程和思路：证明系统可以在某时间范围内维持数据可用是一个更好的策略。这中间采取的方式和手段都是为这个目标服务的。通过关注于目标，而不是具体手段，我们可以避免落入“过程全部成功，系统却仍然存在故障”这样的陷阱。

对灾难预防来说，从“任何部分都可能出错”到“任何部分都一定会出错”是一个巨大的飞跃。通过对所有的可能性以矩阵式排列，可以帮助我们确保数据完整性保障过程能够覆盖所有可能的所有组合—这可以让我们睡一晚上的好觉。只有将整个恢复计划持续运行，不断演习才能保障我们在其他的364天里也能睡上好觉。

当能做到在N时间范围内恢复数据之后，我们可以通过更迅速的、更细致的数据丢失检测来降低N，最后的目标一定是将N趋近于0。接下来，我们可以将重点从恢复转为预防，目标一定是保障全部数据随时可用—当你做到了这一点之后，每天去海边沙滩上睡也都不是问题了。

第27章 可靠地进行产品的大规模发布

作者：**Rhandeev Singh**、**Sebastian Kirsch**、**Vivek Rau**

编辑：**Betsy Beyer**

Google这样的互联网公司比起传统公司来可以更快地发布和迭代新的产品和新的功能。SRE在整个流程中起到的作用是确保快速迭代不会影响到网站的稳定性。为此SRE还建立了一个专门的团队——发布协调小组，负责为具体的开发团队提供技术顾问和支持。

这个小组同时维护一个“发布检查列表”，包括针对每次发布需要检查的常见问题，以及避免常见问题的手段。这个列表在实践中被证实是保障发布可靠性的重要工具。

让我们来讨论一个普通的Google服务—Keyhole。该服务负责给Google Maps以及Google Earth提供卫星图像。在平常的一天里，Keyhole 需要每秒响应几千条图像请求。在2011年圣诞节前夜，突然之间流量暴涨到25倍日常峰值—接近了每秒100万条。是什么导致了此次流量暴涨呢？

圣诞老人要来了！

几年前，Google和NORAD（北美空防指挥中心）合作推出了一个圣诞节主题的网站，上面显示了圣诞老人在全球各地发送礼物的记录，并且允许任何人跟踪圣诞老人的行踪。（该网站是半娱乐性质的，允许用户自己在上面添加自己收到的礼物记录和地理位置，实际上收集了很多数据以提高地图的精确性）。网站的一个重要功能是“飞行模式”，利用卫星图像来实时跟踪圣诞老人的飞行路线。

虽然这样的一个网站是半娱乐性质的，但它的发布具有一切难度高和危险性强的特点：

1.不可改变的截止日期（Google可不能因为网站没有上线而不过圣诞节）,2.大规模宣传活动，3.数百万的使用人群，4.流量增速非常快

（所有人都会在平安夜浏览这个网站）。可不要小看几百万个小孩同时刷新网站——这个项目有潜力能将Google的全部服务搞垮。

Google SRE团队为了这次发布做了很多基础设施的改动，以确保圣诞老人发送礼物的过程不受影响。由于没有一个人想目睹小朋友们无法使用网站而伤心，我们甚至将产品内嵌的几个保护性功能开关命名为“让小孩子哭吧”。SRE中的一个特殊团队“发布协调工程师”（LCE）负责预测发布可能出现的问题，以及在不同的开发小组之间协调工作。

发布一个新产品或者一个新功能对每个公司来说都是一场考验——几个月，甚至几年的努力都是为了向世界宣布这一刻。传统公司的发布周期很长，而互联网公司的发布周期是非常不同的。快速发布、快速迭代变得简单的原因是他们仅仅需要在服务器端发布，而不需要发布到每个使用者的电脑上。

Google将发布定义为用户可见的应用程序代码更新。取决于每次发布的特点——属性以及时间，发布步骤的多少，以及发布复杂程度——发布的流程也很不一样。按照这个定义来说，Google有时可以做到每周发布70次。

这种发布速度要求我们创建和维护一个精简的发布流程。一个每三年发布一次新产品的公司不需要详细的发布流程——下次发布时，任何之前创建的发布用的组件可能都过时了。这样的公司也不可能有足够的机会创建出一个好的发布流程，因为无法通过反复实践来提升这个流程的速度和可靠性。

发布协调工程师

Google的软件工程师在编码和设计上的经验很丰富，同时也对产品的细节非常了解。但是，他们对同时发布给几百万用户的挑战不是很了解——在这个过程中还要最小化故障的可能性，以及最大化产品的性能指标。

Google通过建立一个SRE内部的专职顾问团队来帮助解决这些问题。该团队由软件工程师和运维工程师组成，还包括一些曾经有过其他

SRE团队经验的人。该团队负责指引开发者构建符合Google标准的可靠的、可扩展的、稳定的，而且性能一流的产品。这个团队中的人被称为发布协调工程师（Launch Coordination Engineering,LCE）,LCE团队通过以下几个方面确保发布流程平稳：

- 审核新产品和内部服务，确保它们和Google的可靠性标准以及最佳实践一致，同时提供一些具体的建议来提升可靠性。

- 在发布过程中作为多个团队之间的联系人。

- 跟进发布所需任务的进度，负责发布过程中所有技术相关的问题。

- 作为整个发布过程中的一个守门人，决定某项发布是否是“安全的”。

- 针对Google的最佳实践和各项服务的集成来培训开发者，充分利用内部的文档和培训资源来加速开发。

LCE团队的成员会在整个服务生命周期的不同阶段进行审核（audit）工作。大部分审核工作是在新产品或者新服务发布之时进行的。如果某个团队在没有SRE支持的情况下发布产品，LCE会介入提供咨询服务确保发布平稳进行。如果某项产品已经有了很强的SRE支持，LCE团队经常也会在关键功能发布时介入帮助。发布一个新产品所面临的挑战经常和平时稳定运行一个可靠性很强的服务是完全不同的（SRE团队的强项在于后者）。LCE团队因为参与过几百次发布因而经验很丰富，他们可以在服务进行SRE审核的时候提供帮助。

发布协调工程师的角色

我们的发布协调工程师团队由直接招聘的工程师和其他有经验的SRE组成。LCE的技术要求与其他的SRE团队一样，并且需要有很强的沟通和领导能力—LCE需要将分散的团队聚合在一起达成一个共同目标，同时还需要偶尔处理冲突问题，并且为其他工程师提供建议和指导。

专职的发布协调小组提供了以下优势。

广泛的经验

作为一个真正的跨产品小组，发布协调小组的全部成员都积极参与到公司全部产品线活动中。丰富的跨产品知识和与很多团队的良好关系使得LCE团队是公司中最适合进行知识共享的媒介。

跨职能的视角

LCE对发布过程具有整体视角，这使得该团队可以协调多个分散的小组，包括SRE、研发团队和产品团队等。这种全局视角对复杂发布流程来说更重要，这些发布通常需要协调七八个不同时区的团队共同完成。

客观性

作为一个中立的建议方，LCE在SRE、产品研发组、产品经理和市场运营团队之间起到了平衡与协调的作用。

由于LCE 是SRE的一部分，LCE也非常注重可靠性。如果其他公司不像Google这么关注可靠性，可能会采用另外一种组织架构。

建立发布流程

Google已经用超过10年的时间打磨发布流程，在这段时间中我们发现了好的发布流程具有的一些特征。

轻量级

占用很少的开发时间。

鲁棒性

能够最大限度地避免简单的错误。

完整性

完整地、一致地在各个环节内跟踪重要的细节问题。

可扩展性

可以应用在很多简单的发布上，也可以用在复杂的发布过程中。

适应性

可以适用于大多数常见的发布（例如在产品界面上增加新的UI组件），以及可以适应全新的发布类型（例如Chrome浏览器和Google Fiber的第一次上线）。

正如你想的那样，这些需求是互相有冲突的。例如，想要设计一个流程来同时满足轻量级和完整性是很困难的，在这些参数中取舍平衡需要持续不断地投入。Google成功地采用了以下几种手段来达到目的：

简化

确保基本信息正确。不需要为所有的可能性做准备。

高度定制

有经验的工程师会针对每次发布定制流程。

保证通用路径快速完成

识别出几类发布流程所具有的共同模式（例如在新的国家发布产品），针对这类发布提供一个快速简化通道。

经验证明，工程师会绕过那些过于烦琐，或者附加值不高的流程——尤其是在产品上线的压力之下——整个发布流程可能被视为另外一个拦路虎。正是由于这样，LCE必须持续不断地优化整个发布的体验，在成本与收益上保持平衡。

发布检查列表

检查列表可以用来减少失败，并且在多个职能部门之间保证一致性和完整性。常见的例子包括民航起飞之前的检查列表以及手术之前的检查列表（参见文献[Gaw09]）。同样的，LCE使用发布检查列表来评估每次发布过程。该发布列表（参见附录E）可以帮助LCE工程师评估发布

过程，并且给发布团队提供具体的待办事项，以及更多信息的链接。以下是列表中的一些例子。

- 问题：是否需要一个新的域名？

—待办事项：与市场部门协调想要的域名，并且去申请注册。通过此链接向市场部门提交申请。

- 问题：是否存储持久化信息？

—待办事项：确保实现了备份，通过此链接了解备份实现的细节问题。

- 问题：该服务是否有可能被某个用户滥用？

—待办事项：在服务中实现限速和用户配额管理。通过此链接了解有关的共享服务。

在具体实践中，我们可以提出近乎无限的问题，这样会导致该列表无限增长。LCE需要精心挑选合理的问题，确保开发者负担在可控范围之内。为了控制该列表的增长，向列表中增加新的问题需要经过公司副总裁的批准。LCE遵守以下原则：

- 每一个问题的重要性必须非常高，理想情况下，都必须有之前发布的经验教训来证明。

- 每个指令必须非常具体、可行，开发者可以在合理的时间内完成。

这个检查列表需要持续不断地投入精力进行维护，以确保里面的问题仍然有效：里面的建议可能会过期，内部系统可能会改变，之前关注的重点可能由于新的策略和系统改变而过时了。LCE会持续维护该检查列表，如果发现问题会进行小型修改。每一年或者半年，会有专人评估整个检查列表，如果发现过时的项目，会与对应的服务负责人和领域专家共同解决问题。

推动融合和简化

在大型组织中，工程师可能不完全了解一些针对常见任务的现有的技术设施（例如限速功能）。如果缺乏足够的引导，这些工程师经常会重新发明轮子。将常见的功能融合于一套通用的基础设施类库可以避免这种场景，收益非常明显：消除了重复劳动，使得各个服务之间的知识更容易传递，同时可以确保这些基础设施有足够的关注，以此提高它们的工程质量和服务质量。

几乎所有的Google团队都参与一个通用的发布流程，这使得发布检查列表可以作为一个载体来推动通用基础设施的融合。LCE会推荐现有的基础设施作为服务的基础构建单元—这些基础设施服务已经经过了多年的优化和加固，可以帮助消除容量、性能和扩展性方面的不确定性。常见的基础设施，包括限速功能和用户配额，服务器数据推送功能，新版本发布功能等。这种标准化机制可以大幅简化发布检查列表：例如，关于限速功能要求的长篇大论可以简化为一句话“利用系统X实现限速功能”。

由于LCE对Google所有的产品都有很丰富的经验，因此他们也是识别可简化区域的最佳人选。在执行某次发布过程中，他们会收集第一手资料：哪些方面对发布造成的困难最多，哪些步骤花费的时间超多，哪些问题不停地反复用类似的方法解决多次，可以用通用基础设施服务来替代，或者基础设施服务中哪些地方存在重复。LCE有很多方法来加速发布过程，他们也会替产品团队争取权益。例如，LCE可能会和某个审批流程复杂的团队一起合作简化他们的审批流程，同时为常见场景实现自动化审批。LCE同时可以向基础设施组提出要求，作为用户和服务负责人之间的沟通桥梁。由于之前多次发布的经验，LCE的话语权更重，可以确保建议和意见受到重视。

发布未知的产品

当某个项目进入某个新的产品空间，或者是垂直领域的时候，LCE可能需要创建一个全新的检查列表，这通常需要引入相关的领域专家。当起草新的检查列表时，检查列表应该关注一些宽泛的主题，例如可靠性、故障模式和流程等。

例如，在发布Android之前，Google几乎没有处理大规模消费者产

品运行不能直接控制的客户端代码的经验。虽然我们可以通过推送新版本的JavaScript代码在数小时，甚至数天内修复Gmail服务的一个Bug，但对移动设备来说这却是不可能的。因此，负责移动端发布的LCE需要和移动端领域的专家一起确定现有的检查列表内容是否适用，以及是否需要添加新的项目。在这个对话过程中，一定要记住确保每个问题的“目的”与待发布产品的设计密切相关，而不能简单地增加无关问题。负责这类非常规的发布的LCE必须要坚守安全发布的抽象原则，再具体细化成新的检查列表内容，确保他们对开发者来说是有用的。

起草一个发布检查列表

检查列表是可靠发布新服务和新产品的重要组成部分。我们的检查列表随着时间不停地变长，但是LCE团队会周期性地调整内容。具体的检查列表细节每个公司都会不同，因为这些具体事项必须跟公司内部的服务和基础设施相关。在接下里的几节中，我们会从Google LCE 检查列表中抽取一些主题来讨论。

架构与依赖

针对系统架构的评审可以确定该服务是否正确地使用了通用基础设施，并且确保这些基础设施的负责人加入到发布流程中来。Google拥有很多内部服务，它们经常作为新产品的构建组件。在接下来的容量规划过程中（参见文献[Hix15a]），依赖列表可以用来保证该服务的相关依赖都有足够的容量。

示范问题：

- 从用户到前端再到后端，请求流的顺序是什么样的？
- 是否存在不同延迟要求的请求类型？

待办事项：

- 将非用户请求与用户请求进行隔离。
- 确认预计的请求数量。单个页面请求可能会造成后端多个请求。

集成

很多公司的对外服务都要运行在一个内部生态系统中，这些系统为如何建立新服务器、配置新服务、设置监控、与负载均衡集成，以及设置DNS配置等提供了指导。这些内部的生态系统通常随着时间迁移而不断变大，经常包括自己的特质与陷阱。所以以下这些内容每个公司都会不同：

- 给服务建立一个新的DNS。
- 为服务配置负载均衡系统。
- 为服务配置监控系统。

容量规划

新功能通常会在发布之初带来临时的用量增长，在几天内会消除。这种尖峰式的负载或流量分布可能与稳定状态下有显著区别，之前的压力测试可能失效。公众的兴趣是很难预测的，有些Google产品需要为预计容量提供15倍以上的发布容量。首次发布时限制在单独区域或者国家内有助于建立以后大规模发布时的信心。

容量规划与冗余度和可用性都有直接关系。例如，如果需要三个相互复制的部署点来服务100%的峰值流量，那么我们就需要维护4个或者5个部署点，这其中包括1或2个冗余节点，这样可以让用户不会受到数据中心维护或者其他突发情况的影响。数据中心和网络资源通常需要很长的准备时间，需要足够提前申请才能获取到。

示范问题：

- 本次发布是否与新闻发布会、广告、博客文章或者其他类型的推广活动有关？
- 发布过程中和之后预计的流量和增速是多少？
- 是否已经获取到该服务需要的全部计算资源？

故障模式

针对新服务进行系统性的故障模式分析可以确保发布时服务的可靠性。在检查列表的这一部分中，我们可以检查每个组件以及每个组件的依赖组件来确定当它们发生故障时的影响范围。该服务是否能够承受单独物理机故障？单数据中心故障？网络故障？如何应对无效或者恶意输入，是否有针对拒绝服务攻击（DoS）的保护？如果某个依赖组件发生故障，该服务是否能够在降级模式下继续工作？该服务在启动时能否应对某个依赖组件不可用的情况？在运行时能否处理依赖不可用和自动恢复情况？

示范问题：

- 系统设计中是否包括单点故障源？
- 该服务是如何处理依赖系统的不可用性的？

待办事项：

- 为请求设置截止时间，防止由于请求持续时间过长导致资源耗尽。
- 加入负载丢弃功能，在过载情况中可以尽早开始丢弃新请求。

客户端行为

在传统网站中，很少需要将用户的合理滥用行为考虑进来。当每条请求都是由于用户行为（例如单击链接）触发的，请求的速率会受到用户单击速度的限制。双倍的负载需要双倍的用户来产生。

这个原则当考虑到某些客户端在没有用户输入的情况下执行操作的时候就不适用了，例如，手机客户端APP周期性地将数据同步到云端，或者使用了自动刷新功能的网站等。在这几种场景中，客户端的滥用行为很容易影响到服务的稳定性（防止爬虫和拒绝服务攻击是另外一个话题，与设计正常用户的安全行为不同）。

示范问题：

- 该服务是否实现了自动保存/自动完成（输入框）/心跳等功能？

待办事项：

- 确保客户端在请求失败之后按指数型增加重试延时。
- 保证在自动请求中实现随机延时抖动。

流程与自动化

Google鼓励工程师们使用标准工具来自动化一些常见流程。然而，自动化永远不是完美的，每个服务都有需要人工执行的流程：构建一个新版本，迁移服务到另外一个数据中心中，从备份中恢复数据等。为了保障可靠性，我们应该尽量减少流程中的单点故障源，包括人在内。

这些剩余的流程应该在发布之前文档化，确保在工程师还记得各种细节的时候就完全转移到文档中，这样才能在紧急情况下派上用场。流程文档应该做到能使任何一个团队成员都可以在紧急事故中处理问题。

示范问题：

- 维持服务运行是否需要某些手动执行的流程？

待办事项：

- 将所有需要手动执行的流程文档化。
- 将迁移到另外一个数据中心的流程文档化。
- 将构建和发布新版本的流程自动化。

开发流程

Google是版本控制系统的重度用户，几乎所有的开发流程都和版本控制系统深度整合。很多最佳实践的内容都围绕着如何有效使用版本控制系统而展开。例如，我们大部分的开发都是在主线分支上（mainline）进行的，但是发布版本是在每个发布的分支上进行的。这种方式使得在分支上修复每次发布的Bug更简单。

Google还利用版本系统做一些其他的事情，例如存放配置文件等。版本控制系统具有许多优势—跟踪历史、修改记录，以及代码审核—也都适用于配置文件。在某些案例中，我们也会自动将版本配置系统中存放的配置文件自动推送到生产环境中，工程师只需要提交一个修改请求就可以自动发布到线上。

待办事项：

- 将所有的代码和配置文件都存放到版本控制系统中。
- 为每个发布版本创建一个新的发布分支。

外部依赖

有时候某个发布过程依赖于某个不受公司控制的因素。尽早确认这些因素的存在可以使我们为它们的不确定性做好准备。例如，服务依赖于第三方维护的一个类库，或者另外一个公司提供的服务或者数据。当第三方提供商出现故障、Bug、系统性的错误、安全问题，或者未预料到的扩展性问题时，尽早计划可以使我们有办法避免影响到直接用户。在Google产品发布的历史上，我们曾经使用过过滤/重新代理服务、数据编码流水线，以及缓存等机制来应对这些问题。

示范问题：

- 这次发布依赖哪些第三方代码，数据、服务，或者事件？
- 是否有任何合作伙伴依赖于你的服务？发布时是否需要通知他们？
- 当我们或者第三方提供商无法在指定截止日期前完成工作时，会发生什么？

发布计划

在大型分布式系统中，很少有能够瞬间完成的事件。就算能够做到，为了保障可靠性，这样快速发布也并不一定是好主意。复杂的发布过程可能需要在不同子系统上单独启动各个功能，每个配置更新都可能

需要数小时才能完成。在测试实例中可以工作的配置文件并不能够保障一次性被推送到生产实例上。有的时候为了成功发布，可能要进行一个复杂的操作过程或者编写特殊的代码来保障流程的正确性。

市场宣传部门或PR部门经常会提出他们的要求，使得整个流程更复杂。例如，某个团队可能需要在主题演讲进行时启用某个功能，但需要在主题演讲之前屏蔽掉。

备用方案是发布计划的另外一个方面。如果没有在主题演讲中成功启用这个功能怎么处理？有的时候这些备用方案可能就是简单地准备另外一页幻灯片，“我们会在接下来的几天内发布该功能”，而不是“我们已经发布了这个功能”。

待办事项：

- 为该服务发布制定一个发布计划，将其中每一项任务对应到具体的人。
- 针对发布计划中的每一步分析危险性，并制定对应的备用方案。

可靠发布所需要的方法论

正如本书其他部分描述的那样，Google在多年稳定运行系统中研发了一系列方法论，其中的某些方法论非常适用于安全发布产品。这些方法论可为你在日常运维中提供很多优势，但是在发布阶段就实践它们也是很重要的。

灰度和阶段性发布

系统管理员常说的一句谚语是“永远不要在正在运行的系统上做改动”。任何改动都具有一定的危险性，而任何危险性都应该被最小化，这样才能保障系统的可靠性。在小型系统上测试成功的改变，在Google这种全球分布式、高度复制的系统中很难直接使用。

Google的发布很少是“立即可用”的——在某个具体时间后整个世界都可使用。Google研发了一系列发布模式，用它们逐渐地发布产品和服务

可以降低风险（更多信息可参见附录B）。

根据某个事先定义的流程，几乎所有的Google服务的更新都是灰度进行的，在整个过程中还穿插一些校验步骤。新的服务可能会在某个数据中心的几台机器上安装，并且被严密监控一段时间。如果没有发现异常，服务则会在某个数据中心的所有机器上安装，再次监控，最后再安装到全球全部的服务器上。发布的第一阶段通常被称为“金丝雀”—这和煤矿工人带金丝雀下矿井检测有毒气体一样。通过使用这些“金丝雀”服务线上流量，我们可以观察任何异常现象的发生。

“金丝雀”测试是嵌入到很多Google内部自动化工具中的一个核心理念，包括那些修改配置文件的工具。负责安装新软件的工具通常都会对新启动的程序监控一段时间，保证服务没有崩溃或者返回异常。如果在校验期间出现问题，系统会自动回退。

灰度式发布的理念甚至可以应用在那些运行在非Google机器上的软件。新版本的Android APP可以用灰度方式发布，新版软件可以先发布给一部分用户。新版本覆盖的百分比通常逐渐随时间增长到100%。这种类型的发布在新版本会带来新的后端流量时非常有用。使用灰度发布，我们可以在灰度发布新版本的时候关注其对服务器的影响，以便更早地检测问题。

功能开关框架

Google经常会使用发布前测试之外的一些补充策略来避免故障的发生。创建一个可控更新的机制允许我们在真实负载情况下观察系统的整体行为，用工程力量和时间来换取一定的可靠性保障是很划算的。在逼真的测试环境由于某些原因无法构建时，或者复杂发布中存在不可预知的时候这些机制就非常有用了。

还有，不是所有的改动都可以一样对待。有时我们仅仅是想检查某个界面上的改动是否能提升用户感受。这样的小改动不需要几千行的程序或者非常重量级的发布流程。我们可能希望同时测试几百个这样的改动。

最后，有时候我们想要知道是否一小部分用户会喜欢使用某个新功能，就通过发布一个简单的原型给他们测试。我们不希望花费数个月的时间来优化一个没人想要使用的功能。

为了满足上述这些场景，一些Google产品加入了功能开关框架。某些框架被设计为可以将新功能逐渐发布给0%～100%的用户。每当产品增加这个框架时，该框架都会被仔细调优，以便未来大部分的功能上线时不需要LCE再参与。这种框架通常需要满足以下几个要求：

- 可以同时发布多个改动，每个改动仅针对一部分服务器、用户、实体，或者数据中心起作用。

- 灰度式发布到一定数量的用户，一般在1%～10%之间。

- 将流量根据用户、对话、对象和位置等信息发送到不同的服务器上。

- 设计中可以自动应对新代码出现的问题，不会影响到用户。

- 在严重Bug发生，或者其他副作用场景下可以迅速单独屏蔽某个改变。

- 度量每个改变对用户体验的提升。

Google的功能开关框架基本可归类为以下两类：

- 主要面向用户界面修改的。

- 可以支持任意服务器端和逻辑修改的。

对用户界面修改来说，最简单的功能开关框架是一个无状态的HTTP重写器，运行在前端服务器之前，只对某些Cookie起作用，或者是其他的某种HTTP请求/回复的属性。某个配置机制可以将新代码和一个标识符关联起来，同时实现某种黑名单和白名单机制（例如Cookie哈希取模之后的某个范围）。

有状态的服务一般会将功能开关限制为某个已登录用户的标识符，

或者某个产品内被访问的实例，例如文档ID、某个表格、存储物件等。有状态的服务一般不会重写HTTP请求，而是通过代理或者根据需求转发到其他服务器上的手段来测试复杂功能或者新的业务逻辑。

应对客户端滥用行为

最简单的客户端滥用行为是某个更新间隔的设置问题。一个每60s同步一次的新客户端，会比600s同步一次的旧客户端造成10倍的负载。重试逻辑也有一些常见问题会影响到用户触发的行为，或者客户端自动触发的行为。以一个目前处于过载状态的服务为例，该服务由于过载，某些请求会处理失败。如果客户端重试这些失败请求，会对已经过载的服务造成更大负载，于是会造成更多的重试，更多的负载。客户端这时应该降低重试的频率，一般需要增加指数型增长的重试延迟，同时仔细考虑哪些错误值得重试。例如，网络错误通常值得重试，但是4xx HTTP错误（这一般意味着客户端侧请求有问题）一般不应该重试。

故意的或者不故意的自动请求的同步性会造成惊群效应（正如第24章和第25章描述的那样），这是另外一个常见的滥用行为的例子。某个手机APP开发者可能认为夜里2点是下载更新的好时候，因为用户这时可能在睡觉，不会被下载影响。然而，这样的设计会造成夜里2点时有大量请求发往下载服务器，每天晚上都是如此，而其他时间没有任何请求。这种情况下，每个客户端应该引入一定随机性。

其他的一些周期性过程中也需要引入随机性。回到之前说的那个重试场景下：某个客户端发送了一个请求，当遇到故障时，1s之后重试，接下来是2s、4s等。没有随机性的话，短暂的请求峰值可能会造成错误比例升高，这个周期会一直循环。为了避免这种同步性，每个延迟都需要一定的抖动（也就是加入一定的随机性）。

服务器端控制客户端行为的能力也是一个重要工具。对一个安装在设备上的APP来说，这种控制可能意味着客户端周期性地与服务器联系，并且下载一个配置文件。这个文件可能会启用或者禁用某些功能或者调整参数，例如多久同步一次和重试的频率等。

客户端配置文件甚至可以启用全新的用户可见的功能。通过将新功

能对应的代码在激活之前提前发布到客户端，我们可以显著降低该发布的危险性。发布新版本也能变得更简单，原因是不需要针对功能启用与维护多个并行的发布轨道。这对发布一系列有着各自不同的发布时间的独立功能来说就更重要了，否则需要维护各种组合版本。

通过加入这种功能使得在问题出现时，中止发布更容易。我们可以简单地将该功能关闭，修复代码，再发布新版APP。如果没有这个设置，就需要去掉这个功能，制作一个新版本，再发布到所有人的设备上。

过载行为和压力测试

过载情况是一个复杂的故障模式，因此需要额外加以注意。意料之外的成功经常是某个服务发布时造成过载的最常见因素，其他原因包括负载均衡问题、物理机故障、同步客户端行为、外界的攻击等。

在简化的模型中，假设物理机的CPU用量与某个服务的负载线性相关（例如，请求的数量或者处理的数据量等），一旦可用CPU耗尽，处理过程就会变慢。但是不幸的是，真实的服务很少按照理想模式运转。大部分服务在没有负载的情况下都会变慢，一般是由于各种各样的缓存因素，例如CPU缓存、JIT缓存，以及其他的数据缓存等。随着负载上升，经常有一段时间CPU用量和负载线性同步增长，响应时间基本保持固定。

到达某一点后，很多服务在过载之前会进入一个非线性转折点。在最简单的案例中，响应时间会上升，造成用户体验下降，但是不一定会造成故障（然而依赖服务的速度变慢可能会造成RPC超时，而造成某个用户可见的错误）。在极端情况下，服务会在过载情况下进入完全死锁状态。

这里引用一个具体的负载行为：某个服务会在收到后端错误时记录调试信息。但是由于记录过程会比处理后端成本更高。当该服务进入过载状况，某些后端请求超时时，服务会花更多的CPU时间在记录调试信息上，造成更多的请求超时，最终进入一个完全死锁状态。对运行在Java　JVM上的服务来说，这种类似的死锁状态有的时候被称为“垃圾回

收死亡螺旋”。在这个场景中，虚拟机内部的内存管理会运行得越来越频繁，不停地想要回收内存，直到大部分CPU时间都被内存管理部分占用了。

不幸的是，从理论上很难预测某个服务的过载反应。因此，压力测试是非常重要的，不管是从可靠性角度还是容量规划角度，压力测试对大多数发布来说都很重要。

LCE的发展

在Google的早期时代，研发团队的数量每年会翻倍，该状况持续了几年。这造成工程部门被分割成很多很小的团队，分别负责各自的实验性新产品和新功能。在这种情况下，新手工程师经常会重蹈覆辙，尤其是在发布新功能和新产品的过程中。

为了减少这种重复问题的发生，利用以往的好经验，一小部分有经验的工程师—“发布工程师”自愿组建了一个顾问小组。这些发布工程师为新产品的发布起草了检查列表，包括以下一些主题：

- 什么时候需要跟法务部门咨询。
- 如何选择域名。
- 如何注册域名，正确配置DNS。
- 工程设计和生产部署中的常见问题。

发布工程师的咨询实践被称为“发布评审”，逐渐成为一个很多新产品在发布几天或者几周前需要进行的标准流程。

在两年之内，检查列表中的部署要求部分变得很长很复杂。再加上Google内部部署环境复杂度的上升，做到安全发布某个修改对普通工程师来说越来越难。同时，SRE团队正在快速增长，某些没有经验的SRE经常过于小心，甚至反对任何改动。Google当时正处于双方僵持不下的争执中，产品发布速度受到了影响。

为了改善这种情况，SRE在2004年组建了一个小型但是全职的LCE

团队。该团队负责加速新产品和新功能的发布过程，同时利用SRE理念来保障Google持续发布高可用、低延迟的可靠产品。

LCE负责确保发布过程执行迅速，并且服务不会出现故障。在某个发布过程出现问题时，不会影响到其他产品。LCE同时负责保障内部相关团队都能清楚地知道为了提高上线速度所走的一些捷径，以及所带来的风险。该团队的顾问环节后来被标准化，被称为“生产评审”（production review）。

LCE检查列表的变迁

随着Google的内部环境变得更复杂，LCE检查列表（参见附录E）和发布数量也在增多。在三年半的时间内，一个LCE工程师通过检查列表处理了 350次发布。该团队当时平均只有5个工程师，这就意味着Google在三年半的时间内发布了近1500次。

虽然LCE检查列表上的每个问题都很简单，但每个问题的来源和对应的答案都很复杂。为了充分理解这种复杂度，LCE新员工大约需要6个月的时间进行培训。

随着发布数量的增加，跟上Google每年增长一倍的工程师团队，LCE不停地寻找简化评审环节的方法。LCE将一些发布归类为低危险性的发布，这些发布造成问题的可能性非常小。例如，某个功能发布不需要新的二进制文件，同时流量增长在10%之内，就会被认为是低风险的。这些发布通过一个非常简单的检查列表就可以发布，而其他的高风险发布需要更完整的检查。根据2008年的统计，30%的评审结论是低风险发布。

与此同时，Google的环境也在不断扩展，很多发布的限制都不再重要了。例如，YouTube的收购强迫Google扩展了网络，并且更合理地使用带宽。这意味着其他很小的产品都能“塞入缝隙中”，这样就不需要复杂的网络带宽规划和准备，也就使发布更快了。Google同时开始构建大型的数据中心，以便将多个互相依赖的服务放在一起。这种发展也简化了依赖多个现有服务的新产品的上线速度。

LCE没有解决的问题

虽然LCE一直努力将评审中的官僚主义减少，但是做得还远远不够。2009年时，在Google内部发布小型新服务的难度已经众人皆知。而那些规模非常大的服务面临着各自特殊的问题，LCE也不能解决。

扩展性的改变

当产品大幅超出早期预测而非常成功时，用量会增长两个数量级以上，这就要求进行很多设计变更。这种扩展性上的改变，加上不停增加的功能，经常使产品变得更复杂，更易坏，运维非常困难。到了一定时刻，原始的产品结构已经无法维护，需要重新进行架构设计。进行这种重构，并且将用户从旧架构迁移到新架构上来需要大量的研发和SRE资源，会造成这段时间内的新功能发布受到影响。

不停增加的运维压力

当某个服务发布之后进行运维时，运维压力—人工性和重复性的运维工作一般来说会不断增长，除非有专门的团队来控制这种压力增长。自动化通知太多，部署流程太复杂，以及手工维护工作的增加会消耗越来越多的运维力量，使得团队用于开发新功能的时间越来越少。SRE内部宣传的目标是保障运维工作小于上限50%（参见第5章）。维持运维压力在这个限度以下需要持续不断地跟踪运维工作的来源，同时投入力量来减少这些压力。

基础设施的改变

如果底层基础设施（例如集群管理系统、存储、监控、负载均衡、数据传输）被开发团队不断修改，运行在之上的各服务的负责人需要投入大量精力不停地跟上这些改动的节奏。随着某个基础设施功能的废弃和被某个新功能所替换，服务负责人必须要不停地修改配置文件，或者重新构建可执行文件，也就是“逆水行舟，原地不动”。这个问题的解决方案是指定和执行某种策略，禁止基础设施工程师发布非向后兼容的功能改变，除非他们可以自动化地将客户端迁移到新功能上。在制作新功能的时候就创建这种自动化迁移工具可以减少服务负责人的迁移压力。

解决这些问题需要全公司一起投入，已经大幅超出了LCE的范畴：更好的平台级API和框架（参见第32章），持续构建和自动化测试，以及Google生产环境中更好的标准化和自动化环境组合起来才能解决。

小结

快速增长、产品和服务修改迅速的公司可能会从类似LCE这样的角色中受益。这样一个团队对希望每年或者每两年将开发团队翻倍的公司来说更是很有价值的。公司如果想要将服务扩展到上亿用户，在快速更改下保障可靠性，LCE也很有价值。

LCE团队是Google在快速改变中保障安全性的解决方案。本章介绍了我们特有的LCE角色在10年中积累的一些经验，希望这篇介绍对面临类似问题的其他组织有所启发。

[1]2014年夏天，Mikey离开Google成为第一个US Digital Service部门的主管（https://www.whitehouse.gov/digital/united-states-digital-service）。该部门的一大职责是将SRE理念引入到其他美国政府IT部门。

[2]寻呼机Pager，最原始的报警通知设备。

[3]Prometheus是一个开源时间序列数据库，详细信息请参见http://prometheus.io。

[4]Google是美国公司，所以varz的发音是“var-zee”。

[5]Borgmon的复数形式还是 Borgmon，这和sheep 一样。

[6]Global实例用于汇总集群实例的监控信息，通常运行两个对等的实例是因为全球实例（逻辑）也要运行在某一个集群（物理）中，如果只运行一个，可能会受到单点故障的影响。

[7]很多非SRE团队（研发团队，Google很多软件项目并没有SRE参与）使用一个自动生成器来生成Borgmon配置文件，他们发现这些生成器要比手动维护Borgmon配置文件容易得多。

[8]很多应用程序使用自定义协议来暴露内部状态。例如，OpenLDAP使用cn＝Monitor来暴露内部信息。MySQL使用SHOW VARIABLES命令汇报状态，Apache 使用mod_status 模块。

[9]参见http://golang.org/pkg/expvar。

[10]第2章提到的Borg名称解析系统（BNS）。

[11]见第6章提到的针对现象和针对原因的报警。

[12]12小时这个神奇的数字既能保障在线排错时有足够的历史数据，也能避免内存占用量过大。

[13]这里为了节省空间，没有写出service和zone标签，实际返回是有的。

[14]通过计算变化率的总和（sum of rates）而不是总和的变化率（rate of sums），可以防止结果受到实例重启，或者收集失败等因素的干扰。

[15]虽然varz变量没有指定类型，但绝大部分变量都是简单的计数器。Borgmon的速率函数可以自动检测和正确处理所有计数器重置的现象。

[16]当计数器发生重置的时候，可能短时间看起来counter的值有减少趋势，但是总体来看，一定呈上涨趋势。

[17]service和zone 标签被省略了。

[18]由于标签不完全相符。

[19]10m保证返回前10分钟内的所有值，以这种形式计算可以让规则计算变得更稳定，不会因为某次收集失败而立即认为整个任务都失败了，从而触发报警。

[20]service和zone标签被省略了。

[21]这一章文字的早期版本曾经出现在；login:（2015年10月，第40期，第5篇）。

[22]SRE和产品研发部门之间的天然矛盾，参见第1章。

[23]参见第28章，使用通用故障排查过程手段和方法论往往是学习一个系统的有效方法。

[24]参见https://en.wikipedia.org/wiki/Hypothetico-deductive_model。

[25]例如，第10章提到的，系统暴露的监控变量。

[26]由Theodore Woodward，马里兰州立大学医学系，19世纪40年代提出。可参见https://en.wikipedia.org/wiki/Zebra_（medicine）。值得注意的是，在某些系统中某一类问题可能全被排除了。例如，在细心设计的集群文件系统实现中，由于某个磁盘出现问题导致延迟问题是非常不可能的。

[27]奥卡姆剃刀原理（Occam's Razor），可参见https://en.wikipedia.org/wiki/Occam%27s_razor。但是需要注意的是，系统中可能同时存在多个问题，尤其是有的时候是因为系统中存在一系列低危害性问题，联合起来，才可以解释系统目前的状态。而不是系统中存在一个非常罕见的问题，同时造成了所有的问题现象，参见https://en.wikipedia.org/wiki/Hickam%27s_dictum。

[28]参见https://xkcd.com/552。

[29]例如，没法从理论上解释为什么美国计算机科学专业的PHD的毕业数量在2000年到2009年与人均芝士消耗量曲线相关度极高（r^2＝0.9416）。更多详情可参见http://tylervigen.com/view_correlation?id＝1099。

[30]可以让提交错误报告的人查阅文献[Tat99]，这样他们能够提供更高质量的问题报告。

[31]小心虚假相关的可能性！

[32]在很多情况下，这和著名的“Five Whys”理论很像，可参见文献[Ohn88]。

[33]PCRE在计算某些正则表达式的时候可能需要指数型时间。RE2

可在https://github.com/google/re2下载

[34]文献[All15]中提到这是一个在解决问题中经常被使用的手段。

[35]使用实时聊天或者共享文档可以记录你具体操作的时间点，在事后总结时非常有用。同时这个信息也共享给了其他人，他们可以了解目前事态的进展，就不需要打断你来了解情况。

[36]参见本章后面“神奇的负面结果”一节。

[37]关于如何对系统建模可参看文献[Mea08]，关于寻找单一故障源而忽视系统及其运行环境的方法的局限性，可参看文献[Coo04]和[Dek14]。

[38]参见https://cloud.google.com/appengine。

[39]我们压缩和简化了这个案例，便于读者理解。

[40]虽然带着一个未能确认的Bug上线并不是理想情况，很多时候消除所有已知Bug是不可行的。有的时候，我们只能依靠良好的工程师判断力来尽力消除风险。

[41]数据存储（datastore）查询利用一个索引机制来加速比较，但是一个常见的基于内存的实现经常由for循环组成。如果物件数量较少，线性时间复杂度并不是什么问题，但是在大量数据物件的情况下，延迟和资源使用率都会上升。

[42]BIOS：系统内置的，在操作系统（OS）加载之前运行的计算机程序，负责初始化硬件。

[43]本章的早期版本曾发布在；login：上（2015年4月，第40期，第2篇）。

[44]更多细节请参看http://www.fema.gov/national-incident-management-system。

[45]参见http://www.google.com/policies/privacy/。

[46]如果你想收集事后总结，Etsy发布了Morgue（http://github.com/etsy/morgue）用来管理事后总结。

[47]Google Peer Bonus计划允许工程师给同事发一笔奖金，以奖励他们的杰出贡献。

[48]对这次事件的更详细讨论可参见第13章。

[49]举例来说，给Bigtable做某个改动可能需要大量的工程力量，但是只对某个具体故障提供一些缓解性帮助。但是，如果这项改动可以为其他很多不同的生产故障提供同样的缓解性帮助，那这项改动的回报率可能就很高了。

[50]一方面来说，“造成了很多服务问题”可能是提高服务质量和降低报警数量的起始点。另一方面来说，也可能是由“过于敏感的监控机制”造成的。这些监控常常是由一小撮客户基于不正确的SLO标准造成的。同时，服务问题的数量本身，也不能说明修复某项服务的难度到底有多大，或者本次故障的影响有多大。

[51]本章解释了如何从最大化测试中获得好处。一旦一个工程师针对某个系统制定了一个标准的、合理的测试，剩下的工作是所有SRE团队共同的，这样就可以被视为共享基础设施了。这项基础架构包括一个调度器（用来在不同的项目中共享某种有限的资源），和一些执行者（用来以沙盒形式执行该测试的二进制文件），这两个基础设施组件可以分别作为普通的SRE支持的服务（就像集群存储系统一样），这里不再赘述。

[52]有关相同性请参看http://stackoverflow.com/questions/1909280/equivalence-class-testingvs-boundary-valuetesting。

[53]参见https://google.github.io/dagger/。

[54]一个常见标准是，变更部属应该以影响0.1%的用户流量的服务容量开始，同时按每24小时增长10倍推进。与此同时，还要考虑到变更部属的地域性分布（第二天按影响1%的用户流量的服务容量部属，第三天为10%，第四天为100%）。

[55]从距离上来说，假设24小时内在1%～10%区间持续部属，K＝86400/（Ln 0.1/0.01）＝37523 秒，即10小时25分钟。

[56]我们这里使用的是“大O计法”来衡量复杂度。更多信息可参看https://en.wikipedia.org/wiki/Big_O_notation。

[57]有关这个主题的更多讨论，我们推荐参考文献[Bla14]，作者是我们以前的合作者Mike Bland。

[58]参见https://github.com/google/bazel。

[59]例如，待测试的代码是一个复杂API调用的包，提供了一个简单以及向后兼容的抽象层。该代码将这个同步的API包装成返回一个future。如果传入的函数调用参数错误，最终执行结果仍然会产生一个异常，但却是在future真正执行的时候才会抛出。这段代码将API执行结果直接还给调用者，测试这段代码时很多参数异常的情况可能都找不到。

[60]本节主要讨论了SRE工具中需要大规模部署使用的。但是，SRE也开发和使用很多不需要大规模部署的工具。这些工具仍然需要测试，但是这些工具不在本章讨论范围内，由于这些工具也有影响最终用户的危险性，所以这里讨论的策略也适用于这些工具。

[61]参见https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey。

[62]参见https://github.com/aphyr/jepsen。

[63]即使某个测试是以同样的随机数发生器种子重复运行的，所有的任务都是按相同顺序被杀掉的，在杀掉任务与伪造的用户流量之间并没有任何序列化保证。因此之前测试过的错误代码路径不一定能再次重现。

[64]参见http://xkcd.com/303/。

[65]可能是通过类似 Amazon Mechanical Turk 这样的服务招聘而来的。

[66]参见https://github.com/google/protobuf。

[67]不是因为软件开发工程师不写这类工具，而是这类工具通常需要纵越不同的技术领域，以及同时横跨多个抽象层，这种类型的工具的

开发团队经验比较少，系统管理员团队较多。

[68]参见https://groups.google.com/forum/#!topic/public-dns-announce/67oxFjSLeUM。

[69]不幸的是，不是所有的DNS解析器都遵守权威服务器的TTL值。

[70]否则，用户为了拿到IP列表，必须要建立一个TCP连接。

[71]例如，参见Doorman（http://github.com/youtube/doorman），提供了一个协作性分布式客户端节流系统。

[72]见Wikipedia,“Positive feedback”,https://en.wikipedia.org/wiki/Positive_feedback。

[73]软件看门狗机制（watchdog）常常以一个定期唤醒的线程实现，这个线程会检查上次醒来之后是否完成了任何工作。如果没有完成任何工作，该线程会假设服务器已经卡住，而主动杀掉自己。例如，假设某种特定请求会定时发送给该服务器，如果没有收到或者没有成功处理该请求则意味着有可能某组件已经失败 有可能是软件服务器自己，或者发送请求的组件，或者是中间连接的网络。

[74]通常由于地理分布因素，这并不是一个很好的假设。见第2章“任务和数据的组织方式”小节。

[75]作为留给读者的练习：编写一个模拟器，观察在不同的负载限制值和重试次数下，后端所能完成的有价值工作的数量变化。

[76]有的时候，你会发现服务容量的一大部分都来自于缓存中的服务，如果无法访问缓存，就无法服务那么多请求。同样的观察也适用于延迟：缓存层可以帮助你实现延迟目标（通过从缓存中提供服务以降低平均延迟），但是在没有缓存的情况下就不可能达到。

[77]Kyle Kingsburg 写了很多有关分布式系统正确性的文章，里面包含了这类数据库中会发生的意外和不正确的系统行为，可参见https://aphyr.com/tags/jepsn。

[78]值得注意的是，最初的 Paxos 算法性能确实差强人意，但是已

经在近年来得到了极大的提升。

[79]本文曾在ACM Queue（2015年3月，第13期，第3篇）中发表。

[80]这里不会讨论每个具体任务自身原因导致的失败。

[81]Wikipedia：解析、变形、加载，参见http://en.wikipedia.org/wiki/Extract,_transform,_load。

[82]Wikipedia：大数据，参见http://en.wikipedia.org/wiki/Big_data。

[83]Jeff Dean的讲座“构建大型分布式系统的软件工程建议”是一个非常好的资源（参见文献[Dea07]）。

[84]流式系统设计，参见http://en.wikipedia.org/wiki/System_Prevalence。

[85]分布式系统的MVC模式是一个从Smalltalk借鉴来的松散比喻，原始比喻是用来描绘图形用户界面的设计结构的（参见文献[Fow08]）。

[86]Wikipedia:MVC，参见http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller。

[87]原子性、一致性、隔离和持久性，可参见https://en.wikipedia.org/wiki/ACID。MySQL、PostgreSQL等SQL数据库是满足这些条件的例子。

[88]基本上可用、软状态和最终一致性，可参见https://en.wikipedia.org/wiki/Eventual_consistency。Bigtable和Megastore 是BASE系统的例子，它们经常被称为“NoSQL”。

[89]有关ACID和BASE的更多讨论，请参见文献[Gol14]和[Bai13]。

[90]大二进制对象，Binary Large Object，可参见https://en.wikipedia.org/wiki/Binary_large_object。

[91]参见https://en.wikipedia.org/wiki/Zero-day_（computing）。

[92]黏土板是人类已知的保存最久的文字记录形式。针对数据长期保存的讨论，请参见文献[Con96]。

[93]当阅读到这条建议时，读者可能会问：既然我们需要在数据存储之上提供一套API来实现软删除功能，为什么不更进一步提供针对更多的误删除场景的恢复支持呢？从Google的经验出发，以Blobstore为

[94]例：Blobstore的API实现了很多安全功能，包括默认的备份策略（离线副本），端到端校验机制，以及默认的软删除期限。实践证明，在多起事故中，软删除机制都帮助用户避免或者降低了数据丢失的额外难题的严重性。虽然还有很多删除保护机制值得一提，但是纵观全局，软删除机制是在防范Bug和误删除场景中最有效的机制。

[95]“快照”在这里指的是对存储实例的一个只读、静态的视图，如SQL数据库的一个快照。快照通常使用“copy-on-write”技术实现，以提高存储效率。快照成本来自两个方面：第一，它们与在线数据存储挤占存储容量。第二，数据修改速度越快，copy-on-write的作用越有限。

[96]有关GFS类型的复制信息可参见文献[Ghe03]。有关Reed-Solomon 纠错码机制的更多信息，可参见https://en.wikipedia.org/wiki/Reed-Solomon_error_correction。

[97]参见http://en.wikipedia.org/wiki/Catch-22_（logic）。

[98]在实践中，编写代码对大部分SRE来说并不是什么难事，因为大部分SRE都是很有经验的软件开发者。这种要求也使得SRE非常难以招聘。从这个案例以及其他案例中可以了解为什么SRE坚持需要招聘熟练的软件开发工程师（参见文献[Jon15]）。

[99]在我们的经验里，云计算工程师经常拒绝针对生产环境中的数据删除设置报警，因为删除速度随着时间波动很大。然而，这种报警的关注点其实应该在全局范围，而非局部。比起针对每个用户的删除速度的报警，更有用的报警是全局汇总过后的删除速度，阈值可以是一个比较极端的数字（例如10倍于观察到的95% 值），这样的报警规则有利于检测到极端情况。

第IV部分 管理

本书最后的一些主题涵盖了团队内部合作以及团队之间协作的话题。任何一个SRE成员都不能脱离团队，而SRE在这方面有一些比较值得讨论的经验。

任何一个想要建立有效SRE团队的组织都需要格外关注培训。一个用心设计、用心执行的培训课程，可以培训SRE在复杂和快速变动的环境下有效应对。这样的课程可以在新员工刚入职的数周或数月内将多年积累的最佳实践和最佳策略一次灌输给他们。请见本书第28章中的详细描述。

任何一个参与运维的人都知道，运维大型服务通常会带来很多中断性任务：生产环境出现问题，需要更新二进制文件，不停地咨询请求等。在这样多变的环境下有效地管理这些中断性任务是SRE必备的技能，详情请见第29章。

如果这种多变的状况持续过久，SRE团队需要从运维过载的情况中恢复过来。我们在第30章中详细描述了一套应对流程。

在第31章中，我们讨论了SRE团队中的不同角色；跨小组、跨地域、跨大洲的沟通问题；如何组织进行生产环境评审会议；以及SRE内部协作的一些成功案例。

最后，在第32章中，我们讨论了SRE运维工作中的核心：生产环境部署评审（PRR），这个评审过程是接入新服务的关键步骤。我们讨论了如何组织PRR会议，如何在这个成功的但是有局限性的模型上更进一步。

Google SRE的其他有关资料

构建一套可靠的系统需要一系列技能的混合，从常见的软件开发到不那么常见的系统分析，还包括软件工程学。我们在“SRE：系统工程学”（参见文献[Hix15b]）中讨论了这种工程学。

在“SRE招聘”（参见文献[Jon15]）中，我们讨论了如何招聘到合格的SRE，这是组建一个高效团队的关键。Google的招聘方法在Work Rules!（参见文献[Boc15]）[1]等文章中也有体现，但是招聘SRE有其特殊的地方。就算以Google的招聘经验来看，SRE候选人也比一般的工程师要难以寻找，更难以进行有效的面试。

第28章 迅速培养SRE加入on-call

如何给新手带上喷气背包，同时保证老手的速度不受影响？

作者：**Andrew Widdowson**

编辑：**Shylaja Nukala**

新的SRE已经招聘到了，接下来怎么办

假设已经招聘到了新的SRE雇员，接下来我们必须要在工作中培训他们。在工作初期的教育与技术培训上投入足够的力量，有助于将新加入的SRE培养成更好的工程师。合适的培训计划可以使他们更快地进入工作状态，同时保证他们的工程技能更加平衡与可靠。

成功的SRE团队离不开信任—为了维持全球化服务的正常运转，我们必须信任on-call团队了解系统如何运行，[2]可以诊断系统的异常情况，善于利用资源和寻求帮助，以及可以在压力下保持镇静快速解决问题。那么，仅仅考虑“新手需要学习哪些知识才能参与on-call”是不够的，根据上述需求，我们还需要考虑以下问题：

- 现有的on-call工程师如何能够评估新手是否已经准备好参加on-call？

- 我们如何能够更好地利用新员工带来的激情与好奇心，使老员工也能从中获利？

- 组织哪些活动不仅可以丰富团队知识，还能够获得大家的喜爱？

作为一个学员，每个人的学习习惯都不一样。首先我们要认清的是，新招聘到的学员一定是学习习惯各有不同，如果只针对某一种学习习惯制定培训计划肯定是不够的。因此，没有一种教育方式是最优的，也没有一种教育方式是适用于所有SRE团队的。表28-1列出了Google SRE熟悉并推荐的一些培训方式（以及对应的注意要点）。这些培训课程是Google SRE内部多种培训课程的一个代表，通过它们，团队可以保障自己拥有足够的知识，以及将这些知识传递下去。

表28-1:SRE培训课程

推荐的培训方式	错误的培训方式
设计一个具体的、有延续性的学习体验，以便学员跟进	通过给学员安排一些烦琐的工作（处理警报 / 工单）来培训
鼓励反向工程，利用统计学来思考问题，以及多思考问题本质	要求严格按照现有的操作过程、检查列表，或者手册来执行命令进行训练
鼓励学员分析失败的案例，分享好的事后总结来阅读	将故障掩盖起来，以便躲避指责
创造一些受控的，但是逼真的场景让学员利用真实的监控环境和工具来修复	在学员加入 on-call 之后，第一次遇到问题时才会去尝试修复
在团队内以角色扮演的形式演习理论上可能发生的问题，让大家在这个过程中交流彼此的解决问题的方式	在团队中将知识隔离起来，创造出一些只在某个领域内的专家
给学员创造条件让他们参与见习 on-call，和实际轮值的 on-call 工程师交流经验	在学员还没有对服务有全面认识的情况下，就要求他们成为主 on-call
让学员与 SRE 老手一起共同修订培训计划中的某个部分	认为 on-call 培训素材是静态的，非专家不可更改
帮学员一起找到一个具有一定复杂度的项目，帮助他们在整个技术栈内建立自己的地位	将新项目全部分配给 SRE 老手，新手 SRE 只能做一些零工

本章的其他部分讨论了我们在培养SRE过程中发现的最有效的几种方式。这些主题在图28-1中进行了展示。

这张图内包含了SRE团队可以采用的几个培训新员工的最佳实践，同时保证老员工不会落伍。从下面描述的很多个工具中，我们可以从中选择最适合团队的几个来使用。

这张图中包含两个轴：

● X 轴代表工作类型的范围，从抽象的，一直到具体的工作。

● Y轴代表时间。从上至下，新加入的SRE通常对系统和服务知识了解很少，阅读之前故障的事后总结对他们很有帮助。新的SRE也可以试图从基本元素出发反向工程整个系统，因为他们本来对系统也没有任何了解。当学员对系统有一定程度的了解，并且已经进行了一些实际操作时，SRE就可以参与见习on-call了，同时可以修订文档中不完善或者过时的部分。

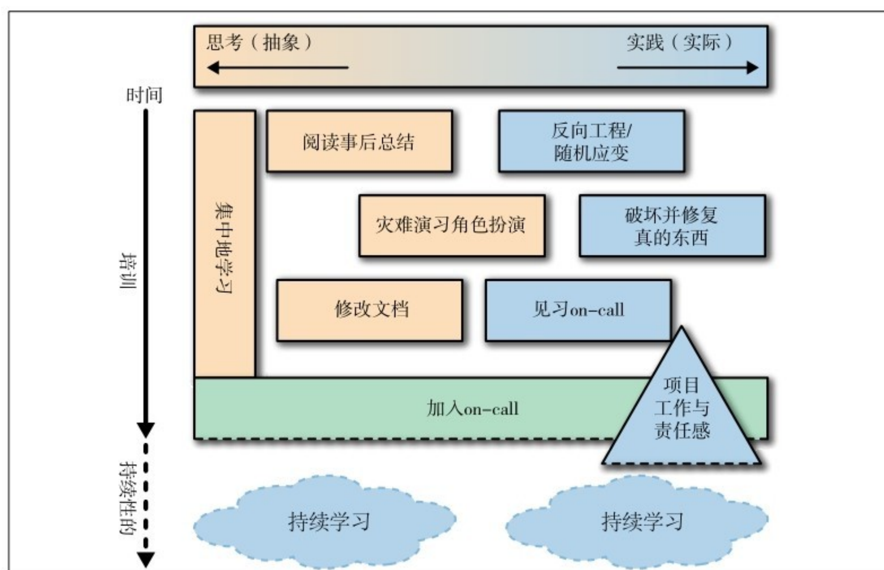


图28-1：培养SRE加入on-call的计划图

阅读上述图表的一些建议：

● 正式参加on-call是SRE新员工的一个重要里程碑，在这之后，学习过程将基本靠自己主导，是未定义的—所以SRE参与on-call之后的培训计划都是用虚线标注的。

● 三角形的“项目工作”意味着新项目刚开始都很小，随着时间推移，复杂程度逐渐增大，所需投入也在增加，很有可能在参与on-call之后还会继续。

- 这里的某些活动是抽象性的，还有些是被动性的。其他的一些活动，则是非常具体和非常具有主动性的。还有一些活动是混合型的。这样的安排是为了满足不同学习方式的人，让他们都能找到适合自己的活动。

- 为了使培训效果最优，培训课程应该节奏合理：有些可以立即开始，有些则应该在SRE正式参见on-call之前开始，而其他的则应该是持续性的，同时要求SRE老员工也参与的。在SRE正式参加on-call之前，应该处于持续不断的学习过程中。

培训初期：重体系，而非混乱

正如本书其他部分所描述的那样，在SRE团队的职责中，主动性任务[3]和被动性任务[4]兼有。每个SRE团队都应该具有的一个重要目标是利用积极主动的办法去减少和限制被动性工作的产生，在培训的过程中也应该体现出这一点。下面提到的是一个非常常见，但是比较差的培训经历：

John 刚刚加入了 FooServer SRE团队。这个团队的老手们每人都负担了很多琐事，例如回复工单、处理报警，以及进行琐碎的二进制文件发布等工作。John入职的第一天，所有新的工单就全部指派给了他。同时，主管告诉John可以寻求任何一个有经验的SRE成员的帮助来了解每个工单的背景知识，以便处理。“当然了，你会有很多东西要学习，这个只能靠你自己了”，主管说道，“最后你一定会学会快速处理这些工单的。某一天你就会突然开窍，发现你已经掌握了我们全部的工具，全部的流程，以及全部的系统知识。”这时候一个老成员补充了一句，“我们这里都是摸着石头过河的。”

这种“浴火重生”式的学习方式一般都反映了团队的目前状况；关注于运维，被动性的SRE团队一般采用这种被动式的教育方式来“培训”新员工。如果幸运的话，新招来的、习惯在未知中探索的工程师最后能够从这种大坑里爬出来。但是更常见的是，很多很有能力的工程师都没法在这种环境中成长。虽然这种培训最终可能能够培养出一些不错的运维工程师，但实际效果肯定也不会太好。这种培训方式同时假设该团队的

大部分知识都可以靠“动手实践”来获得，而不是靠逻辑推理得出。如果这个职位只需要处理工单就够了，那么这个职位其实并不是一个SRE职位。

SRE 学员肯定会存在以下问题：

- 我现在正在做的是什么工作？
- 我现在的进度如何？
- 什么时候我能积累到足够的经验参加on-call？

新的SRE成员通常刚从另外一个公司跳槽，或者是刚毕业，或者刚从传统的软件工程师或系统管理员职位转换成没那么清晰的SRE职位，这已经足够打击他们的自信了。对很多比较内省型性格的成员来说（尤其是面临上面的问题2和3时），培训过程中的混乱或者不确定性会导致他们学习速度变慢，甚至无法适应。所以，我们应该考虑使用下一节所描述的方法。这些建议与处理工单和报警有同样的效果，但是更加系统化、阶段化，也更加适合培训。

系统性、累积型的学习方式

在系统中加入某种顺序性，以便新的SRE成员可以建立某种学习路径。任何类型的系统性培训都要比直接处理随机出现的工单和琐事要好，但是我们应该有选择地将一些理论性和实践性的学习组合到一起：一些新成员经常会用到的、系统的抽象概念应该优先，而学员也应该尽早进行真实的实践操作。

了解任何一个技术栈，或者任何一个子系统都需要一个起始点。我们应该考虑将培训按照类似作用分组，还是按照常用的执行顺序分组。例如，如果团队负责的是一个实时、直接面向用户的服务系统，可以按以下课程顺序进行培训。

1）请求是如何进入系统中的

网络和数据中心的一些基本概念，前端的负载均衡系统，代理等。

2）前端服务

应用程序前端，日志记录，用户体验SLO等。

3）中层服务

缓存，后端负载均衡系统。

4）基础设施

后端，基础设施，计算资源管理等。

5）整体

调试的一些技巧，问题升级的流程，紧急情况的演练。

究竟如何实际教授这些课程（可以是非正式的白板讨论，也可以是正式的培训，以及实际操作练习等）是需要团队主管以及负责设计和进行培训的SRE讨论的。Google 搜索SRE团队利用一个“on-call学习检查列表”来组织自己的培训资源。简化版的检查列表可能如下所示。

搜索结果混合服务器（Mixer）	
前端：frontend server 需要调用的后端：Result Retrieval Server、Geolocation Server、Personalization Database SRE 专家：Sally W、Dave K、Jen P 开发团队联系人：Jim T、results-team@（邮件列表）	必备知识 • Mixer 部署在哪些集群里 • 如何回滚 Mixer 更新 • 哪些后端是 Mixer 的“关键路径”，以及原因
阅读并理解以下文档： • 结果混合服务总览：“请求执行”章节 • 结果混合服务总览：“生产环境”章节 • 生产手册：如何发布 Mixer 新版本 • Mixer 性能分析	**回答以下综合性问题：** • 如果某个常规发布日正好是公司假期，发布时间如何变动 • 如果新推送的 Geolocation Database 数据有误，如何修复

注意，上述章节并没有直接将流程、调试步骤以及手册嵌入其中；而是将专家的联系方式罗列出来，指出最有用的文档资源，以及一些服务的基本知识，最后还提供了一些简单的问题以便自检。该文档同时提

供了一些实际的操作性环节，这样学员可以知道他们完成这个检查列表可以学到哪些知识和技能。

最好能够让所有参与的人都能大概了解某个学员究竟记住了多少知识。这种反馈机制的建立虽然不一定用正式的考试来进行，但是最好能包括一些问答形式的家庭作业。培训讲师通过检查学员的答案可以了解他们目前的学习进度，以及是否应该进入下一阶段。关于服务内部工作原理的问题和下面列出的比较类似：

- 哪些后端服务器是“关键路径”中涉及的，以及为什么？
- 该服务的哪些方面可以简化，或者自动化？
- 你认为该架构的第一个瓶颈在哪里？如果该瓶颈确实出现问题，哪些步骤可以缓解问题？

可以考虑在服务访问权限控制配置中实现一种分层模式。第一层访问权限允许学员只读访问组件的内部信息。接下来则允许修改生产环境状态。随着检查列表项目的完成，学员将会逐步拥有系统的更高权限。搜索SRE团队将这称为“升级”，[5]所有的学员最后都会被赋予系统最高权限。

目标性强的项目工作，而非琐事

SRE是天生的问题解决者，所以我们应该给予他们一个难题去解决！ 在学习过程中，允许新成员向整个服务中加入一点点新东西是很有激励性的。这也是鼓励团队之间构建信任的好方法，因为老员工需要和新员工进行交流，了解新的组件或者新的流程。在Google内的标准做法是：所有的工程师都会被分配一个初始项目，给他们提供足够的基础设施知识，同时让他们可以为服务做一点小的但是有意义的贡献。让新的SRE成员将时间同时分配在学习和项目工作中可以给他们一种参与感和效率感，这比让他们专攻两者中的任意一个要好。常见的新手项目类似于：

- 在服务技术栈中增加一个小的，但是用户可见的功能修改，接下来跟随这个修改一直到发布到生产环境中。了解开发环境的配置和二进

制文件的发布流程可以确保他们和开发团队联系密切。

- 针对现有的服务监控盲点增加新的监控。新手需要了解监控逻辑，同时将这些异常情况与系统知识相结合。

- 选择一个还没有被自动化的痛点进行自动化，使得新的SRE可以给团队成员减轻一些日常负担，从而受到团队的欣赏。

培养反向工程能力和随机应变能力

我们可以提出一系列“如何”培训SRE的意见，但是究竟应该培训SRE“什么”呢？具体的培训材料当然要取决于工作中需要用到的技术，但是这里关注的更重要的问题是：我们究竟需要培养什么样的工程师？在SRE这种复杂度和规模下，工程师仅仅做到关注运维、传统的系统管理员模式是不够的。不光要有大规模海量工程化思维，SRE同时也应该具有如下特点：

- 在日常工作中，他们会遇到从未见过的系统，必须要具有很强的反向工程能力。

- 在海量规模下，很多异常情况都很难检测，他们必须具有统计学知识，用统计学而不是流程化的方式去发现问题。

- 当标准的流程不工作时，他们必须能够随机应变，解决问题。

接下来我们会详细讨论这些特点，这样我们才能知道如何培训SRE来获取这些知识和技能。

反向工程：弄明白系统如何工作

工程师对他们从未见过的系统都很好奇—工作原理是什么。通过对系统工作原理的基本理解，同时愿意深入研究调试工具、RPC框架，以及系统的二进制文件来理解其中的数据流动，SRE就能够在陌生环境中更高效地处理始料未及的额外难题。教授SRE成员如何调试和诊断应用程序，同时让他们练习利用这些调试信息进行推断，可以使他们在日后的工作中更习惯使用这种方式工作。

统计学和比较性思维：在压力下坚持科学方法论

我们可以将SRE处理大规模系统紧急情况的方式理解为他们在实时展开的决策树中不断抉择的过程。在紧急情况处理的有限时间中，SRE只能在几百个可能的选择中选择几个动作去执行，缓解故障。因为时间通常是有限的，SRE必须能够有效地在决策树中进行抉择。这种能力的获得通常要靠经验积累，而经验只能通过有效和真实的实践获得。这种经验必须同时和一系列精心构造的假设结合，当这些假设被证实，或者证伪时，可以更进一步地减少决策空间。用另外一种说法，进行系统故障调试的过程很像是一种游戏："下列哪一个东西与其他不同"？这里的"东西"可能是内核版本、CPU架构、二进制文件版本、地区性的流量区别，或者其他几百种因素。从架构层面来讲，该运维团队必须要保证这些因子是可控的，每个因子都应该是进行独立分析和比较过的。然而，我们也应该培训SRE新员工从一入职就成为好的数据分析者。

随机应变的能力：当意料之外的事情发生时怎么办

假设某SRE试着按照手册修复某个问题，但是没有成功。该故障系统的开发团队无法联系，这时候怎么办呢？当然要临场发挥！通过学习多种解决不同问题的工具可以保障工程师有多种手段处理问题。在故障发生时固守陈规而不去做现场分析，可能会导致无法找到问题根源。在非常复杂的故障排除过程中，SRE通常需要利用很多未经验证的假设来进行决策。在SRE培训初期加入一些关于决策"陷阱"的描述，以及关于如何能够及时抽身，从更高的层面换一个角度去思考问题的解决方案，是非常有价值的。

那么，哪些课程和实践可以将新手SRE培养成具有这些特性的高效率人才呢？需要针对上述这些特点因地制宜，因材施教，以下是一个覆盖了上述所有要点的课程例子。

将知识串联起来：反向工程某个生产环境服务

"新SRE需要学习Google Maps生产服务的一部分时，与其让其他人传授知识给他，不如自己动手_利用反向工程手段自己了解服务，让其他人纠正他的错误和补充遗漏的部分。"这样做的结果是——这可能比我亲

自授课效果还要好，虽然我已经为这个服务on-call超过5年了！”

——Paul Cowan,Google SRE

Google内部最受欢迎的一门课程是“如何反向工程某个生产环境服务”。这门课程提出的场景初看起来比较简单：整个Google 新闻团队—包括 SRE、开发工程师、产品管理团队等—都去参加团建了：目的地是百慕大三角区。我们已经30天没有他们的消息了。课程学员作为新组建的Google 新闻SRE团队成员，需要自行搞清楚整个服务端到端的工作原理，以便能够保障它继续运行。

背景介绍过后，学员会经过一系列交互性的、目的性很强的练习，在Google基础设施中跟踪他们浏览器发出的某个请求。在整个过程中的每一段，我们强调学习利用多种方式来发现生产服务之间的连接关系的重要性，这样可以防止某些连接关系被遗漏。在课程过程中，我们还会让学员跟踪另外一个来源请求，这会展示出我们一些最初的假设范围太窄，产生的是错误的结果。接着还会让学员学习通过其他方式了解服务原理。我们利用Google内部高度集成的调试信息，展示互相之间的RPC连接关系，以及现有的白盒与黑盒监控系统来决定用户请求的数据路径。[6]在这个过程中，我们构造了一个系统组件图，同时讨论了学员未来将会经常见到的共享基础设施服务。

在课程末尾，会给每人分配一项任务。每个学员需要和自己团队的资深SRE共同选择未来on-call系统的一部分。利用课程中学到的知识，该学员需要自行构建出整个技术栈的组件图，同时与资深SRE共同讨论。学员肯定会在这个过程中忽略一些小的但是很关键的细节问题，这些恰恰是最好的讨论话题。资深SRE在这个过程中很有可能也会学到一些知识，因为系统不断变化，任何人都可能出现理解上的偏差。整个SRE团队都应该抓住一切机会学习系统的新变化，而最好的方式恰恰是与团队中最新的成员，而不是最资深的成员交流。

有抱负的on-call工程师的5个特点

on-call轮值一般不是SRE的主要目标，但是生产环境运维经常会需要响应某种类型的紧急事故。能够负责任地进行on-call需要该员工对运

维的系统有着一定程度的了解和熟悉，“能够参与on-call”与“了解足够多，并且可以自己随时自学其他的知识”具有相同的意义。

对事故的渴望：事后总结的阅读和书写

“忘记过去的人必然会犯重复的错误。”

——George Santayana，哲学家和散文家

事后总结（参见第15章）是持续改进的一个重要部分。这是找出某次大型事故的根本原因的一种手段，对事不对人。当书写事后总结时，一定要记住这个文档的最佳受众可能是一个还没有入职的工程师。不需要多么特别的编辑过程，Google某些最好的事后总结只需要小小的改动就可以变成学习材料。

但是，这些事后总结如果被束之高阁是没有什么用的。每个团队必须收集，并且整理有价值的事后总结文档，将它们用于未来新手的教育材料。某些事后总结文档就是需要死记硬背的，某些事后总结文档可以作为“课程”进行教授，可以让学员了解大型系统的结构性弱点，或者新奇的故障方式，这些都是非常有价值的。

某个事后总结文档的负责人不仅仅包括作者。很多团队都以能够经历过大型故障，并且文档化故障的解决过程为荣。我们应该将最好的故障文档收集起来，提供给新手学习—同时也可以提供给其他相关团队，以及上下游团队来阅读。同时，也应该要求相关团队将他们的最佳事后总结文档公布出来。

Google的某些SRE团队会举办“事后总结团体学习”，将一些有价值的事后总结文档集中分发、阅读，并组织集体讨论。原始作者经常会被请到这些会议中，其他团队则会举办“故障沙龙”这样的活动，请原始作者以非正式的形式出席，描述故障的过程，并且组织讨论。

定期举办这种事后总结的阅读活动，包括那些触发条件和应对办法，可以潜移默化地构建起SRE的知识库，培养他们理解生产环境的能力和应对问题的能力。事后总结同时也是未来进行抽象化故障演习时的良好材料。

故障处理分角色演习

“每周我们都会有一次会议，指定某个成员来现场处理某个故障场景——经常是Google历史上曾经发生过的故障。”该成员——就像参加电视娱乐节目的选手那样——需要向主持人提出问题，或者告知目前要采取的行动，主持人会回答问题，以及告知该行动的结果。这就像是魔域传奇（早期的互动式魔幻游戏）。SRE成员身陷于各种监控图表组成的迷宫中，必须要解决用户请求延迟过高的问题，或者是防止数据中心崩溃，又或者是防止某个错误的Google Doodle被显示在网页上。

——Robert Kenndy，前Google 搜索SRE,healthcare.gov 网站运维[7]

当面对一个经验各异、能力各异的SRE团队时，如何能够将所有人紧密结合在一起启发他们互相学习是一个挑战。同时，如何将SRE文化—不断解决问题的文化—传授给新员工，同时保证老员工始终能跟上不断变化的环境和服务发展也是一个挑战。Google SRE团队通过一个老传统—“故障处理分角色演习”来解决这些问题。这个活动同时也被称为“命运之轮”（wheel of misfortune）或者“走木板”（walk the plank）等，这些名字对新加入的SRE来说不会那么吓人。

这个活动做得最好的时候，会成为一个全体成员每周共同学习的机会。具体过程很简单，和桌上RPG游戏类似：“主持人”（GM）选择两个团队成员成为主on-call和副on-call；这些SRE与GM同时坐在房间的最前面。主持人宣布某个警报发生了，这个on-call团队需要进行调查和现场处理该报警。

GM事先已经精心准备了一个故障场景。这个场景可能是基于某个以前发生过的故障，但是新成员没有经历过，或者是老成员已经忘记的事故。或者该场景是某个新功能或者马上上线的功能的假想故障场景，这样参与的所有人都不知道如何处理。还有更好的是，某个成员可能发现了一种全新的生产环境问题，而今天的场景就基于此展开。

在接下来的30～60分钟内，主on-call和副on-call要试着寻找问题的根源所在。GM会随着问题的展开而提供更多的信息，例如会告知大家

某个监控图表的走势等。如果该问题需要向其他团队寻求帮助，GM会假扮成其他团队成员回答问题。虚构场景不一定是完美的，所以GM可能需要纠正参与者见到的一些不相关问题，将他们指引到正确的路线上来，同时可能需要提供更清晰的解释，甚至增加一些其他的信息，[8]包括提出一些紧急而非常有方向性的疑问。[9]

当灾难演习RPG运转成功时，每个人都会从中学到一点东西：可能是一个新工具、新技巧，或者解决问题的另外一个角度，或者（对新员工来说）对成功解决某个问题的认可。也许这项练习可以鼓励团队成员面对下周的挑战，甚至成为接下来某一周的GM。

破坏真的东西，并且修复它们

新手能够通过阅读文档、事后总结文档，以及在培训中学到很多SRE的知识。通过故障分角色演习可以让某个新手快速进入状态。然而，最好的经验还是要从动手破坏或者修复真实的生产环境中得来。当新手SRE加入on-call以后，通常会有很多机会进行动手实践，但是真正的学习应该在这之前进行，这样才能有所准备。因此，我们应该在培训早期就引入一些动手实践，使得学员可以更好地使用公司内部的工具和监控系统来解决某个故障问题。

在这些交互过程中，真实性是非常重要的。理想情况下，我们的团队应该有一套多数据中心部署的系统，可以保证我们将流量从至少一个实例切走，将该实例作为练习使用。或者，我们可以搭建一个小型、但是功能齐全的QA实例，可以用作培训。在可能的情况下，最好在这个实例中加入一些人工产生的负载，模拟真实用户流量，以及资源的消耗。

在这样一个虚拟负载的环境下学习，机会是非常丰富的。资深SRE也会遇到很多问题：配置问题、内存泄露、性能下降、请求导致的崩溃以及存储的瓶颈等。在这个逼真的、但是相对风险较低的环境中，培训者可以更改系统的行为，强迫新SRE找到与生产环境的不同点，寻找可能导致问题的因素，并且最终修复系统。

有的时候，让某个资深SRE精心构造一个具体故障场景比较耗时，可以从另外一个角度来进行：从一个正常状态出发，慢慢将环境中某个

组件逼近瓶颈，通过监控系统观察对上下游系统的影响。Google搜索团队非常重视这项活动，将其称为：“一起来摧毁一个搜索集群！”如下所示：

1.作为一个团队，我们讨论哪些性能特点可能会在过程中受到影响。

2.在实际操作之前，我们进行一次调查，让所有参与的人都提出一个针对系统行为变化的假想和背后的逻辑。

3.具体执行操作，验证之前的假设，并且从逻辑上说明为什么系统会出现这样的现象。

这个团体活动，每个季度会进行一次，有助于在生产环境中发现亟待解决的新Bug—系统并不会像我们想象的那样优雅降级。

维护文档是学徒任务的一部分

很多SRE团队要维护一个“on-call学习检查列表”。该列表中包含一系列阅读材料，以及关于系统的各种技术和知识的完整列表，学员必须充分掌握该列表才能进行见习oncall。如表28-1所示，该学习检查列表对不同的人来说有不同的作用。

- 对学员来说

–这个文档可帮助学员了解运维系统的边界。

–通过学习这个文档，学员可以了解系统的重要组件，以及背后的原因。当他们掌握了这些系统的知识之后，可以去了解其他更广泛的知识，而不是纠结于一些可以通过日后学习得来的细节问题。

- 对导师和管理者来说：学员的学习进度可以通过检查列表来反映，检查列表可以回答以下问题：

– 学员今天学习的是哪一节？

– 哪一节是最让人困惑的？

- 对其他团队成员来说：文档成为一种社会契约，只有掌握这个文档的员工才能加入on-call。这个学习检查列表成为一个知识的标准，所有的团队成员都必须符合这个标准。

在快速变化的环境中，文档可能很快就过期了。过期的文档对资深SRE来说并不是什么问题，因为他们会用其他方式跟踪目前情况的变化。新手SRE需要一个更新过的文档，但是却可能没那么自信去修改它。在这个文档中加入适当的结构，可以同时保障新手的积极性，与资深SRE的知识共存，保证所有人的知识都是最新的。

在搜索SRE团队里，我们会在新成员加入之前先组织评审现有的学习检查列表，同时按更新程度将各节排序。随着新成员的加入，他们会被指定更新某一个或者某几个章节。如图28-1所示，我们将资深SRE和开发者的联系方式与对应的组件标记起来。我们鼓励学员尽早与这些专家沟通，这样他们就能够直接学习到相关技术的内部原理。一段时间后，当学员开始熟悉学习列表的规范和格式后，会要求他们提供某一章节的修改部分，这些修改必须经过列在表上的资深SRE的评审才能提交。

尽早、尽快见习on-call

不管进行多少次假想中的演习，学习多少培训材料，也无法完全保证某个SRE能够彻底准备好on-call。到达一定时候，进行实战操作一定会比进行假想演习更为有效。但是，对新手来说必须要能够在接收到第一条真实报警之前进行演习。

当学员掌握了所有的系统基础知识之后（例如，完成on-call学习列表），我们会考虑将报警系统的警报复制一份给学员。一开始，可以让他们只接收工作时间内的报警。可以利用他们的好奇心驱动自己。这些“见习”on-call过程是培训者了解学员进度的好机会，也是学员了解on-call责任的好机会。通过安排这些学生见习团队中的多个成员进行oncall，可以促进团队之间的互相了解和信任，为该成员最终加入on-call做准备。同时，这种方式也有助于资深成员合理安排时间，避免团队出现后劲不足。

当一个报警发生时，新手SRE并不是问题解决的负责人，这样可以让他们在没有任何压力的情况下进行操作。他们现在可以亲眼观察故障发展，而不是在事后学习。学员可以和主on-call共享一个终端，或者仅仅是坐在一起讨论。在故障解决后的某个合适时机，on-call可以和学员一起讨论背后的处理逻辑和过程。这种活动可以帮助见习学员记住更多细节。



如果某个较大的故障发生，书写事后总结比较必要时，on-call成员应该将学员加为共同作者。不要将整个书写过程全部交给学员，因为这可能会导致他误认为事后总结是一项琐事，专门交给新手来完成的：千万不要造成这种印象。

某些团队还会增加一个最终步骤：让有经验的on-call成员“反向见习”学员。学员会成为主on-call，负责处理所有相关的问题，但是资深的on-call成员会在一旁观察，让其独立地检查问题，而不会做任何修改操作。资深SRE同时可以提供积极的帮助，帮忙确认要执行的动作，以及提供必要的提示。

on-call之后：通过培训的仪式感，以及日后的持续教育

随着知识的增多，学员最终可以理解大部分的技术栈，并且具有自主学习其他部分的能力。这时，他们应该加入到服务的on-call轮值中来。有些团队会有一个最终考试，在赋予学员on-call权限和责任之前会再最后测试一次。其他的新SRE团队仅仅要求学员提供证明其完成了学习检查列表即可。不管怎样，加入on-call都是一个里程碑，应该作为一个仪式，团队中的所有人共同庆祝。

在学员加入on-call之后学习就停止了吗？当然不是！为了保持新鲜

知识，SRE团队需要主动、积极地了解不停出现的新改动。整个技术栈的某些部分可能在不知情的情况下被重新架构，或者扩展了，导致团队对这个方面的知识过期。

应该为整个团队建立一系列常规的学习时间，利用这个时间讨论新的和即将发生的改动，最好由推进这个改动的SRE，甚至与开发者共同展示。如果可能的话，可以将这些讲座记录下来，作为未来学员的培训资料。

通过一些努力，我们可以让SRE和开发者沟通得更加密切。也可以通过其他一些方式促进团队学习，例如：让SRE给开发者做讲座。开发团队对SRE工作与挑战越了解，日后工作中达成一致就越容易。

小结

针对SRE培训的投入是非常有价值的，不管是对渴望尽快掌握知识的学员还是对整个团队来说都一样。通过利用本章中描述的一些实践，我们可以更好地培养出合格的SRE，同时可以不断磨炼团队的技能。具体如何将这些实践应用起来，要靠读者自己，但是任务目标是很明确的：作为SRE，我们必须要能够以比扩张机器更快的速度扩张我们的团队。祝读者在创建学习型团队的路上一切顺利！

第29章 处理中断性任务

作者：**Dave O'Connor**

编辑：**Diane Bates**

“运维负载”这个词语，在讨论到复杂系统时，是指维持系统正常运转必须进行的工作。例如，如果我拥有一辆车，我需要对它进行维护操作，例如给车加油，或者其他的常规维护工作，以确保这辆车保持运行状态。

任何复杂系统都和创造这个系统的人一样，都不是完美的。在管理

这些系统造成的运维负载时，一定要记住这一点。

运维负载有许多类型，有一些很明显，一些则不那么明显。具体使用的术语各有不同，但是基本分为三大类：紧急警报、工单，以及其他的持续性运维活动。

紧急警报，是关于生产环境中出现的问题以及相关问题发出的警报，目的在于通知接收人处理紧急情况。有的时候，这些问题是单调重复的，处理起来不需要很多思考。但是，有的时候问题需要详细地在线调查才能解决。紧急警报通常配有预期的响应时间（SLO），一般是几分钟。

工单，是指那些需要进行某种操作以应对客户的需求。和紧急警报类似，工单也可以是很简单的、重复的，几乎不需要临场发挥。简单的工单可能只是需要对某个配置文件进行代码评审。复杂的工单则可能需要处理一个较为复杂，不那么寻常的设计评审，或者容量规划方案评审。工单同样也有SLO，但是它的响应时间一般是数小时，甚至数天数月。

日常运维任务（也被称为接力棒任务，或者琐事，参见第5章）包括团队负责的代码或者配置文件发布，或者是处理临时的但是又紧急的客户需求。虽然这些任务没有指定的SLO，但是这些任务会打断日常的工作。

一些类型的运维负载是可以预计并且计划的，但是更多的负载都来自于未预料的地方，可能会牵扯某个人很长时间，甚至需要某个人现场决策这件事情是否可以稍后处理。

管理运维负载

Google有几种方式管理团队中每种类型的运维负载。

紧急警报主要是通过设置专门的主on-call工程师来处理的。也就是说，让一个工程师独立接收和响应紧急警报，处理发生的事故或者故障。该主on-call工程师可能同时需要响应用户支持请求，包括将问题升级给对应的开发者等。为了减少紧急警报对整个团队的工作的打断，以

及避免出现事故无人负责的问题，Google的on-call轮值每次只有一个工程师值班。如果该工程师对某个问题不够了解，无法处理，他可以给其他团队发紧急警报，寻求帮助。

一般来说，副on-call工程师作为主on-call工程师的替补。副on-call工程师的职责每个小组都有不同。在某些小组中，副on-call工程师的唯一职责是如果在紧急警报无人响应的时候负责联系主on-call工程师。这种情况下，一般来说，副on-call工程师是由另外一个团队的成员担任的。取决于职责的划分，副工程师可能会继续执行日常工作，而不是中断任务。

工单在每个SRE团队内部的管理方式都不一样：可能是主on-call工程师在轮值过程中也处理工单，也可能是副on-call工程师在轮值的时候负责处理工单，或者团队可以指定某个非on-call工程师来处理。工单可能会随机分配给团队中的所有人，也可能是团队成员自己主动去处理工单。

日常运维任务的管理方式就更多了。有时，on-call工程师负责进行版本发布、配置文件修改等。其他时候，每项任务都会分配给临时的某个具体的团队成员。也有的时候，oncall工程师需要在轮值结束后继续执行某个任务（例如某个需要持续多周的发布，或者某个耗时很长的工单）。

如何决策对中断性任务的处理策略

讨论过具体的运维负载管理策略之后，我们应该回过头来讨论一下在制定这些策略的时候哪些因素是必须要考虑到的。SRE团队发现下列指标对制定策略很有帮助：

- 中断任务的SLO，即预期的响应时间
- 排队的中断性任务有多少
- 中断性任务的严重程度
- 中断性任务的发生频率

- 可以处理某种中断性任务的人有多少（某些团队要求成员必须处理一定数量的工单才能加入on-call）

读者可能注意到了，这些指标都是关于如何能用最低人力成本来满足最低响应时间的。计算人力成本和生产力成本是很困难的。

不完美的机器

从某种意义上讲，人类可以被称为不完美的机器。人会感觉无聊，人的处理器（指思维方式）工作原理不清楚，用户界面（指沟通方式）也不太友好，效率也不高。在工作安排中，能够认识到人的这些缺陷，取长补短是最好的。下面我们讨论一些在决策中需要考虑的几个基本理念。

流状态

流状态（flow　state[10]）是一个软件工程行业内被普遍接受、人尽皆知的理念。“在状态里”可以提升生产力，也可以提升创造性，甚至艺术创造性。进入这个状态可以鼓励员工真正地掌握和优化某个他们负责的任务和项目。但是，如果有其他的事情打断他们，他们就会失去这个状态。我们的目标是尽可能让员工在这个状态下工作。

认知流的概念同样适用，如果某项工作需要的技能很少，虽然他们是持续性的，但是同样无法激发创造力。这些任务具有很明确的目标，反馈很及时，控制感很强，同时时间流逝感也很强烈，例如打扫卫生、开车等。

我们可以通过不断进行低技能需求的工作来进入创造性状态，例如玩一个重复性很强的电子游戏。但是更简单的方法是进行某种高技能需求、高难度的任务，正如工程师每天要做的那些一样。有多种方式进入这种状态，但是目标都是一样的。

认知流状态：富有创造性和参与感

这就是每个人都想进入的状态：某个人在解决问题的过程中，充分了解问题的起因和现状，隐约感觉自己可以解决这个问题。这个人受主

观能动性驱动，甚至忘记了时间，尽可能地忽略其他中断性任务。最大化某个成员在这个状态下的时间是非常好的，这个员工将会产生出很强的创造力，成果丰硕。同时，这个员工自己也会更满意自己的工作。不幸的是，很多SRE成员花费了很多时间进入这个状态，但是被各种事情所打断而无法持续导致很沮丧。又或者，这些人没有试着去进入这种状态，而是一直处于不停被打断的模式中。

认知流状态：愤怒的小鸟

每个人都喜欢做自己知道如何处理的事情。事实上，执行这类任务是进入流状态的最明确的途径。某些SRE在on-call值班的时候可以进入这个状态。他们在追寻问题的原因，与其他人协作，可以明显改善系统整体健康程度的过程中感到极大的满足。恰恰相反，有些工程师将on-call当作是中断性任务，紧急警报过多给他们造成了很大压力。他们可能是在进行项目工作的同时兼职on-call，处理中断性事务，这使得他们处于一种不停被打断的工作环境中，压力非常大。

如果某个人可以全职处理中断性任务，中断性任务就不再是中断性的了。进行一些渐进式的系统改进、处理工单、修复问题等突然有了明确的目标、界限，以及清晰的反馈：修复了X个Bug，或者是紧急警报不再发生了。剩下的问题就是一些分心的事情。当专职处理中断性工作时，之前的项目工作反而是分心的事情了。虽然中断性工作可能是短期内感觉很好，但在SRE环境下，员工在项目工作和中断性工作模式下找到某种平衡一般是更好的。每个工程师对这种平衡的要求不一样。这里要注意的是，某些工程师其实不知道合适的平衡点在哪里，或者和你预想的完全不一样。

将一件事情做好

读者这时候可能在思考上述思想如何能指导具体的实践。

接下来的建议是来自于我在Google内部管理数个SRE团队的经验，主要面向的对象是团队管理层。这篇文章并不关心个人的习惯问题—每个人都可以自由安排自己的时间。这里关注的焦点是整个团队如何管理中断性任务，以便确保整个团队可以正常工作，从而使得团队成员都能

成功。

分心指数

工程师能够被分心的方法太多了。例如，某个名字是Fred的SRE 星期一正常上班，他今天不是on-call，也不负责处理中断性事务。很明显，他希望能够进行他自己的项目工作。倒一杯咖啡，带上“不要打扰”的耳机，开始干活，一切正常，对吗？

但是，下列任何一件事情都可能随时发生：

● Fred的团队使用随机工单分配系统，某个需要今天处理的工单被分配给了他。

● Fred的同事目前负责on-call，收到了一个紧急警报。发出警报的组件Fred最熟悉，所以这个同事过来咨询意见。

● 某个用户将某个工单的优先级提高了，这个工单是上周Fred轮值on-call的时候就在处理的。

● 某个已经持续了三四周的发布到Fred的时候突然出现了问题，Fred需要停下手中的事情检查发布的状态，回滚这个发布等。

● 某个用户过来咨询一个问题，因为Fred为人和善。

● 等等。

最后结果是，虽然Fred有一整天的时间分配给项目工作，他还是很容易被分心。他可以通过关闭E-mail、关掉IM，或者其他手段来减少一定的干扰。但是某些干扰是政策造成的，以及一些无法避免的责任造成的。

我们可以说某种程度的干扰是不可避免的，也是有意为之的。这是正确的：每个人都有甩不掉的Bug，同事之间也会有关系的产生与职责的分配。然而，作为团队来说，是有一些管理方式能使更多的员工不受干扰的。

极化时间

为了限制干扰数量，我们应该减少上下文切换（指工作类型、环境等的改变）。某些中断性任务是无法避免的。然而，将工程师当成是可以随时中断、上下文切换没有成本是不正确的。给每次上下文切换加上成本的考虑。在项目工作中，一次20分钟的中断性任务需要进行两次上下文切换，而这种切换会造成数个小时的生产力的丧失。为了避免这种经常性的生产力丧失，我们应该延长每种工作模式的时间，一天甚至半天都可以。这种策略与“挤时间”（参见文献[Gra09]）策略工作得很好。

极化时间意味着当每个人来上班时，他们应该清晰地知道自己今天是否只是做项目工作，还是只是做中断性工作。这意味着他们可以更长时间地专注于手上的工作，不会不停地被那些他们本不应该负责的事情所打扰。

实际一点的建议

如果本章中描绘的模型不适用于你，那么下面列出了一些具体的建议，可以选用。

一般性建议

不论哪种中断性任务，如果中断性任务的量对一个人来说太高，那么应该增加一个人负责。这个概念很显然适用于工单，但是也适用于紧急警报。on-call工程师可以将紧急警报降级为工单，也可以找副on-call来共同处理。

on-call

主on-call工程师应该专注于on-call工作。如果目前紧急警报较少，那么一些可以随时放下的工单，或者其他中断性事务应该由on-call人员处理。当某个工程师on-call一周时，这一周他应该完全排除在项目进度之外。如果某个项目非常重要，不能等待一周，那么这个人就不应该参与on-call。管理层应该介入，安排其他人替代on-call。管理层不应该期望员工在on-call的同时还能在项目上有所进展（或者其他高上下文切换

成本的活动）。

副on-call工程师的责任取决于这些责任的繁重程度。如果副on-call的工作仅仅是给主on-call工程师做后备，那么可以认为副on-call工程师能够在项目上取得一些进展。如果有其他人负责工单等工作，可以考虑将这两个角色合并。如果由于警报数量原因需要副on-call工程师实际帮助主on-call工程师解决问题，那么他也应该专注于中断性任务。

（另外：总有一些清理工作要做，工单数量可能是0，但是总会有需要更新的文档，需要整理的配置文件等。未来的on-call工程师可以从这些活动中受益。他们就不会再来打扰你了！）

工单

如果目前你是随机分配工单给团队成员，请立刻停止。这样做对团队的时间非常不尊重，和让成员尽可能不被打扰的目标背道而驰。

工单处理应该由全职人员负责，同时保证占用合理的时间。如果团队目前的工单主oncall和副on-call都处理不完，那么需要重新架构整个工单的处理流程，保障任何时间都有两个全职员工处理工单。不要将复杂分散到整个团队中去。人不是机器，这样做只会干扰员工，降低他们的工作效率。

持续性的运维工作

尽可能地让整个团队共同负责这件事。例如变更发布、配置文件修改等，如果这项工作有非常清晰的流程定义，那么就没有任何理由让任何一个人专门负责这件事情。定义一个发布管理者的角色来由on-call和其他中断性任务处理者来承担。同时应该进行正式化交接过程—这有一定成本，但是却保障了接下来的人不受打扰。

负责中断性事务，或者不负责

有的时候，某个中断性任务只有某个目前不在值班的成员能够妥善处理。虽然理想情况下，这种情况不应该发生，但是还是偶尔会发生，我们应该尽一切努力避免这种情况。

有的时候员工在非值班时间也会处理工单，因为这样是显得很忙的好办法。这种行为是错误的。这意味着该成员效率不高，他们会影响对于工单数量的统计。如果某个人负责处理工单，但是其他两三个人也来参与，可能管理层就没法知道到底目前的工单数量水平是否可行。

减少中断

如果团队中需要很多人同时进行中断性任务，那么可能这种负载是不能持久的。有一系列方式可以降低整体的工单负载。

实际分析工单

很多工单轮值和on-call轮值就跟轮流射击差不多，对大型团队来说更是如此。当每隔几个月才值班一次的时候，简单补救一下[11]，叹口气，然后就恢复日常工作是很正常的。接下来的人也会如此，这些工单的根源问题没有人去修复。这样做，整个团队不会有任何进展，每个人都会不停地被同样的问题不停地打扰。

工单的处理也应该有个交接过程，和on-call一样。交接过程可以保证每个处理人之间共享状态。有的时候只是简单地交流就可以触发某种好的解决方案的产生，从而降低整体出现的频率。很多团队都会进行on-call交接与紧急警报评审会议，但是没有几个团队也这样对待工单。

我们应该定期进行工单与紧急警报的整理会议，在这个会议上应该讨论出现的每一类中断性问题，看是否能够找到根本原因。如果这个根本原因是可以在合理时间范围内修复的，我们应该在修复根本原因之前，静音（指暂时不再进行临时处理，而是直接修复根源问题）该问题。这样可以给处理中断性事务的人争取一些时间，同时也设置了一个截止期限。

尊重自己，也尊重用户

这条格言主要适用于用户产生的中断性任务。如果工单很多，解决方法烦琐，我们可以使用一些策略来缓解这种问题。

一定要记住：

- 团队应该为自己的服务设置合理的服务水平。

- 将某些事务推回给客户解决是可以的。

如果团队负责解决客户发出的工单，或者其他中断性任务，我们可以创建一些策略来管理团队的负载。策略可以是临时的，也可以是永久的，取决于具体情况。制定这种策略可以在尊重用户与尊重自己之间选择一个更好的平衡点。有的时候策略和代码一样有用。

例如，如果我们负责支持一个经常出问题，但是没有开发者支持的工具，同时又有一小批挑剔的用户非常需要这个工具，我们可以用其他选项达成目标。要衡量花在维护这个系统上的时间，看看是否值得。在某些时候，如果我们无法说服开发者来修复根源问题，可能这个组件并不是那么重要。我们可以考虑将紧急警报职责还给开发者，或者可以宣称该组件退役，用其他组件来替代，或者采用其他任何更有效的解决办法。

如果某个中断性任务的处理非常复杂，耗时很长，但是却不需要SRE具有的高级权限来完成，这时应该考虑将这种请求交给用户自己完成。例如，如果用户需要贡献一些计算资源，我们可以实现准备好代码，或者配置文件的改动模板，同时给用户清晰的指示让他们来完成这些步骤，最后发给SRE评审。一定要记住，如果客户需要某种事情发生，他们应该自己准备好，花一定的精力来做这件事。

这个解决方案的一个问题是要在尊重用户和尊重自己之间寻找一个平衡点。我们的指导思想应该是确保用户发来的请求应该是有意义的、合理的，同时提供了所有需要准备的材料。同样，我们处理这样的请求也应该是非常及时而且有效的。

第30章 通过嵌入SRE的方式帮助团队从运维过载中恢复

作者：**Randall Bosetti**

Google的SRE团队的标准政策要求团队在项目研发和被动式工作之间均匀分配时间。在实际工作中，这种平衡可能会被每天工单数量的不断增加而被打乱，甚至持续数月之久。长时间承担非常繁重的Ops类型的工作是非常危险的，团队成员可能会劳累过度，不能在项目工作上取得任何进展。当一个团队被迫花费过多时间解决工单的时候，他们就会减少花在改进服务上的时间，服务的可扩展性和可靠性肯定会随之变差。

解决这个问题的一种方式是给处于过载状态的团队临时调入一个SRE。调入该团队后，该SRE应该关注于改善这个团队的行事方式，而不是简单地帮助团队清理积压的工单。通过观察团队的日常工作，提出改善性意见，该SRE可以帮助团队用全新的视角来审视自己的日常工作。这往往是团队本身的成员做不到的。

注意，当采用这个方法的时候，只需要调入一名SRE就够了。同时调入两名SRE不一定会产生更好的结果。对那些防御性比较强的团队来说，可能会激发问题的发生。

当你开始构建自己的第一个SRE团队时，采用本章中介绍的方法可以帮助避免团队走入歧途，退化成那种只专注于工单处理的传统运维团队。在决定采用这种嵌入SRE的方式之前，你应该花一些时间来回顾一下Ben Treynor Sloss的介绍中提到的SRE理念和实践经验，同时应该看一下本书的第6章。

接下来的部分是给即将嵌入团队的SRE的一些指导。

第一阶段：了解服务，了解上下文

在嵌入团队的过程中，你的主要工作是清楚地表达团队目前的流程和工作习惯对于该服务的可扩展性有利或者有弊的原因。你应该提醒该团队，日益增加的工单不应需要更多的SRE来处理。SRE模型的目标是仅仅在系统复杂度上升的时候才增加新人。你应该尝试引导团队建立健康的工作习惯，这样能够减少花费在工单上的时间。这与指出该服务目

前还可以自动化或者进一步简化同样重要。

Ops模式与非线性扩展

Ops模式（Ops mode）指的是一种维持服务运行的特定方法。随着服务规模的扩大，各种工作项目也在增加。例如，某个服务随着规模的扩大，需要管理更多的虚拟机。在Ops模式下的团队通过增加更多的管理员来解决这个问题，SRE则恰恰相反。他们通过编写软件系统或者消除系统瓶颈的方法来解决这个问题。这样运维一个服务所需的人数不会与服务的负载同步增加。

处于Ops模式的团队可能认为可扩展性对于他们来说并不重要（“我的服务很小”）。这可以通过参与一次on-call轮值来确定这种判断是否正确，因为可扩展性的确会影响到我们接下来的策略。

如果该服务中最主要的组件的商业价值很高但是部署规模实际很小（只需要很少的资源或者并不复杂），那么我们应该关注于改进团队现在的工作方式，找到那些阻碍他们改善服务可靠性的部分。时刻记住，你的工作是保证服务正常运转，而不是避免警报发生。

另一方面，如果该服务的增长刚刚开始，我们应该关注于对爆发式增长的准备。某个100 QPS的服务可能在一年内变成一个10,000 QPS的服务。

确定最大的压力来源

SRE团队陷入Ops模式的原因是过分关注如何快速解决紧急事件而不是如何减少紧急事件的数量。团队滑入Ops模式的情况通常发生在面临巨大外界压力时，不管这些压力是真实存在的还是想象中的。在你对服务了解足够深刻、可以讨论设计和部署问题之后，你应该花一些时间来根据历史上各种服务中断造成团队压力的程度将其排序。一定要记住，由于团队的历史和视角问题，一些非常小的问题可能其实正在给团队带来很大负担，而他们自己却不知道。

找到导火索

一旦你确定了该团队现有的最大问题之后，下一步就应该关注那些即将要发生的紧急情况。大部分时候，即将发生的紧急情况是以一个新的、没有被设计成自我管理的子系统的形式出现的。其他的来源包括：

知识代沟

在大型团队中，团队成员可能会过度专业化。当一个团队成员过于专业化时，他在on-call时就会由于缺乏广泛的知识而需要帮助。同时，他的过度专业化反而会导致其他团队成员缺乏只有他才了解的关键系统的知识。

SRE直接参与开发的功能变得越来越重要

这些服务通常没有得到和一般的新功能上线所需的同等水平的关注，因为在规模上他们更小并且当时至少有一位SRE支持。

对“未来的一件大事”的过度依赖

团队成员有时会忽略某种问题长达几个月的时间，因为他们相信“马上上线”的新解决方案能够解决该问题，临时措施没有意义。

开发团队和SRE都视而不见的警报

这样的警报经常被定性为暂时性的，但是这些警报仍然会让团队成员分心。要么仔细地将此类警报调查清楚，要么就修改警报规则让它们不再出现。

任何有客户投诉，并且缺乏一个正式的SLI/SLO/SLA 的服务

参考第4章 关于SLI、SLO和SLA的讨论。

任何一个容量规划都是“增加更多的服务器”的服务：“我们的服务器昨天晚上内存不足了。”

容量规划必须具有足够的前瞻性。如果你的系统模型预测到服务器需要2　GB　内存，那么通过一个负载测试（在上一次运行中显示了1.99 GB内存使用量）并不一定意味着你的系统容量是足够的。

事后总结中的待办事项仅仅只有“回滚导致服务故障的变更”的服务

例如，“将流超时改回到60s”，而不是“弄明白为什么有时需要60s来获取我们的宣传视频的第一个MB。”

现有的SRE以“我们什么也不知道，开发者才明白”来应对的关键组件

为了能对某个组件提供合理的on-call支持，我们至少应该了解当它罢工的时候会造成什么样的后果以及需要多快解决。

第二阶段：分享背景知识

在了解团队内部动态和找到痛点之后，你应该通过建立一些最佳实践来为改善状况做准备。例如，建立书写事后总结的机制，辨别琐事的来源，确定每个琐事的最佳解决方案等。

书写一个好的事后总结作为示范

事后总结体现了团队的集体推理能力，不健康的团队进行的事后总结往往是无效的。一些团队成员可能会把书写事后总结看作是一种惩罚，甚至是无用的。虽然你可能很想回顾之前的事后总结，并且给它们一一评论，这样做其实对团队并没有帮助。相反，这种行为可能会推动团队进入防御模式。

不要尝试改正之前的错误，而是应该亲自负责下一次事后总结的书写。在你和团队的工作过程中，肯定会有至少一次事故发生。如果你当时不是on-call工程师，就应该与oncall　SRE合作书写一个非常好的、对事不对人的事后总结。这个文档可以用来向团队解释他们是如何通过永久性修复根源问题而受益的。永久性地修复根源问题长期来看可以减少事故对团队成员的时间的影响。

如同前文提到的那样，你可能会遇到“为什么是我？”这样的回复。这种回复恰恰说明了该团队相信事后总结过程是报复性的。这一态度来源于对坏苹果理论的认同：整个系统工作良好，如果我们摆脱所有的坏苹果以及它们所犯的错误，系统将会一直运行良好。坏苹果理论已经被

实践证明是错误的，在很多行业中，包括那些对安全极为重视的航空行业中，都有明确的证据来证明，参见文献[Dek14]。你应该向团队成员指出这个理论的错误之处。说服其他工程师书写事后总结最有效的措辞是："在任何一个交互关系错综复杂的系统中，错误是不可避免的。我相信，在当时你利用了最正确的信息做出了最正确的决定。我想要你写下在这段时间里每一个时间点你所想的事情，这样我们就能发现系统在哪里误导了你，以及过程中什么地方对人的认知能力要求过高。"

将紧急事件按类型排序

在我们简化的模型中有两种紧急事件：

- 不应该存在的紧急事件。这些紧急事件会导致那些一般被称作Ops工作和琐事（参见第5章）的发生。
- 其他的紧急事件可能会让on-call成员感到压力巨大，又或者会使他愤怒地敲击键盘。这类紧急事件实际上是日常工作的一部分，团队需要通过构建对应的工具来控制压力。

你应该将团队遇到的紧急事件分为琐事的和非琐事的。最后，将这个列表交给团队并且清晰地解释哪些紧急事件应该被自动化，而其他的紧急事件则是运维服务必须承担的工作。

第三阶段：主导改变

保持团队健康是一个持续的过程。正因为此，这不是你可以通过个人英雄主义来解决的问题。为了确保团队在未来可以进行自我调节，我们需要帮助他们建立一个良好的SRE心理模型。



人类非常善于在环境改变时保持心态不变，所以我们应该关注于创

建（或恢复）正确的初始条件，同时需要传授给他们做出健康选择所必须依赖的简单原则。

从基础开始

无法区分SRE与传统的Ops模型的团队，通常也不能清晰地描述为什么该团队的代码、流程或者文化让他们自己感到不爽。我们不应该试着去一一解决这些问题，而是应该从第1章和第6章中概述的原则开始。

你的第一个目标应该是为团队制定一个服务等级目标（SLO）。SLO 非常重要，因为它对于事故的后果提供了量化分析的依据，同时它也提供了某个流程改变的重要性的依据。一个SLO可能是该团队从Ops模式转向一个健康的、长期的SRE模式的最重要的工具。如果研发团队与SRE团队没有达成这个共识，那本章的其他建议就都帮不上忙了。如果团队目前没有SLO，那么请首先阅读第4章，然后召集技术负责人和管理层会议，讨论和确定SLO。

获取团队成员的帮助

你可能非常着急，想要直接修复你所发现的问题，请一定要克制。因为这样做就是在宣扬“做改变是别人的事”这种想法。相反，请采取以下步骤：

1.找到一个可以由某个团队成员完成的有价值的工作。

2.清晰地解释这项工作是如何以一个永久的方式解决事后总结中出现的问题的。就算是一个健康的团队有时候也是比较短视的。

3.自己亲自作为代码更改和文档修订的评审者。

4.在两到三个问题上重复这个过程。

在发现其他问题时，应该将其写入一个错误报告或一个团队的文档，让团队成员提供意见。这样既可以做到信息共享，又可以鼓励其他团队成员编写文档。（书写文档常常是面临最后期限压力时的第一个受害者）。一定要坚持解释你的逻辑推理过程，强调良好的文档会确保团队在一个新的情况中不会重复旧的错误。

解释你的逻辑推理过程

随着团队逐渐复原，逐渐接受你提出的改变之后，你应该转移注意力到解决那些造成运维过载的日常决策中去。做好准备接受团队成员的反对。幸运的话，这种反对可能是类似于“现在，马上解释为什么要这么做。就现在，在每周的生产会议当中。”

如果不够幸运，没有人会要求你做出解释。这时，你应该通过直接解释你的所有决定来绕过这个问题，不管是否有人要求你这样做。当做出解释时，记得要引用你的建议中的那些基础理念。这样做有助于建立团队的心理模型。在你离开团队后，团队应该能够预知你对某个设计或者某个更改的评论。如果你不解释逻辑推理过程，或者解释得不够充分，就可能会造成团队也形成这种懒惰的风气。所以一定要避免。

对某项决策进行充分解释的例子：

●“我反对最新版本的原因不是因为测试结果有问题，而是因为我们对发布所制定的错误预算已经耗尽了。”

● “发布需要能够安全回退，因为我们的SLO非常高。要想达到SLO的要求，平均恢复时间必须非常短，这样在回退之前进行深入的现场调查是不现实的。”

对于某项决策不充分的解释的例子：

●“我不认为每一个服务器自己生成自身的路由配置是安全的，因为我不信。”

这一决策可能是正确的，但是分析论证上是很薄弱的（也没有详细解释）。团队无法知道你心中所想，所以他们很可能模拟你这种不良行为。相反，尝试“[……]不安全，因为在该代码中的一个错误会导致其他服务同时受到影响，同时额外的代码也可能带来减慢回滚的速度的问题。”

●“自动化机制在遇到部署不一致的情况时应该放弃进行。”

和之前的例子一样，这一决策可能是正确的，但是是不充分的。相

反，尝试“[......]因为我们这里的一个简化假设是全部的改变都通过自动化进行，这种情况的发生说明了有什么东西违反了这一规则。如果这种情况经常发生，我们应该找到和消除造成这种变化的根源。”

提出引导性问题

应该提出引导性问题，而非指责性的问题。当你和SRE团队交流时，试着用一种可以鼓励别人思考基本理念的方式来提出问题。这对于你建立这种模型而言是格外有价值的，因为，按照定义，处于Ops模型中的团队会天生拒绝这种逻辑推理。当你花费一定时间解释了对于不同政策问题的理论推理之后，其实就使得该团队更深刻地理解了SRE的哲学。

引导性问题的例子：

- “我看到任务失败的警报经常发生，但是on-call工程师通常什么都不做。这样会对SLO有什么影响？”
- “这个上线过程看起来非常复杂。你知道为什么创建一个新的服务实例需要这么多的配置文件的更新吗？”

引导性问题的反例：

- “这些旧的、停滞的发布是什么情况？”
- “为什么某个组件要做这么多的事情？”

小结

你应该将以下这些本章中列出的宗旨提供给该SRE团队。

- 从技术角度，最好是量化的角度指出团队需要改变的原因。
- 提供一个详细、具体的“改变”作为例子。
- 解释SRE经常采用的“常识”背后的逻辑推理过程。
- 提供以可伸缩的方式来解决崭新情况所必需的核心理念。

你的最后一个任务是书写一份报告。报告中应该重申你的观点、例子和逻辑推理过程。同时，该报告应该向团队提供一些待办事项，来保证他们会实践你所传授的东西。你应该将报告组织成一份检查报告[12]，解释成功路上的每一个重要的决策。

大部分的工作现在已经完成了。虽然你的嵌入式工作正式结束，但仍然应该参与设计评审和代码评审。在未来几个月中持续关注这个团队，确保他们正在慢慢提高自己的容量规划能力、紧急事件处理能力和部署发布过程。

第31章 SRE与其他团队的沟通与协作

作者：**Niall Murphy**、**Alex Rodriguez**、**Carl Crous**、**Dario Freni**、**Dylan Curley**、**Lorenzo Blanco**、**Todd Underwood**

编辑：**Betsy Beyer**

SRE 在 Google的组织架构中的地位是非常特殊的，这影响了SRE与其他组织进行沟通和合作的方式。

首先，SRE所进行的工作，以及这些工作的进行方式存在巨大的多样性。我们有基础设施团队、服务团队和横向产品团队。我们与产品开发团队存在协作关系，包括规模是SRE很多倍的团队和规模大致相同的团队，还有那些SRE就是研发团队的情况。SRE团队成员拥有系统工程或架构能力（见文献[Hix15b]）、软件工程技术、项目管理能力、领导才能，各种行业背景的人都有（参见第33章）。SRE不光只有一个工作模型，实际上我们已经发现了很多可用的配置方式；这种灵活性符合SRE最终务实的特质。

SRE 不是一个令行禁止（C&C）的组织。一般来说，我们的工作至少包括两个方面：对服务SRE或基础设施SRE团队来说，我们要与相应的为这些服务或者基础设施进行产品研发的团队紧密地协作；另外，很明显，我们也需要进行一些常规的SRE的工作。由于我们直接对系统的性能负责，我们与研发部门的关系是很密切的。但是尽管有这样密切的

关系，SRE的实际组织汇报线是自成体系的。今天，SRE主要是花时间来支持每个具体服务，而非跨部门工作。但我们的文化和我们的共同价值观产生出了非常同质化的解决问题的方法，这恰恰是我们所设计的结果。[13]

上述两个事实决定了SRE团队的沟通和协作是日常工作的两个重要维度。数据流可以作为沟通过程的一个恰当比喻：就像数据必须围绕生产流动那样，数据也要围绕SRE团队流动—关于项目的数据，关于服务状态、生产环境状态以及个人状态的数据。团队的最佳运行状态是，数据可靠地从一个感兴趣的团队流动到另一个团队。思考这种流动的一个方法是思考SRE团队与其他团队建立的接口API。和设计一个API一样，好的设计对于有效性是至关重要的。如果API的设计是错误的，后续改正它将是非常痛苦的。

不管是SRE团队内部的协作，还是与产品开发团队协作，这种将API作为契约的概念也很重要。在这两种情况下，团队都要在一个不停改变的环境中进步。由此看来，SRE的协作与其他快速发展的公司中的协作很像。不同的是，这种协作方式组合了软件工程技能、系统工程的专业知识以及SRE带来的生产经验。最佳的设计和最佳的实现往往诞生于生产环境运维与产品研发在相互尊重的氛围中进行的自由讨论。这就是SRE的承诺：一个专门负责可靠性的组织，拥有与产品开发团队相同的技能，以量化的方式不断改善。我们的经验表明，仅仅只是指派一个人关注可靠性而不具有完整的技能集合，是远远不够的。

沟通：生产会议

虽然有大量文献（参见文献[Kra08]）讲述如何组织有效的会议，但是无效的会议还是会偶尔发生，SRE也不例外。

然而，SRE发现有一种会议非常有用，我们称之为生产会议（production meeting）。生产会议是一种特殊的会议。在这个会议中，SRE团队向自己—以及邀请的嘉宾—描述服务的目前状态。这样那些关心服务的人对服务状态的了解程度得到了提高，同时也能提高服务自身的运维质量。一般来说，这样的会议是针对整个服务的；它不直接

关注每个人的状态更新。这个会议的目标是，在会议结束后，每个参会者对服务的状态了解程度达到一致。生产会议的另一个主要目标是，通过将在生产中获取的知识应用到服务中来以便改善服务。这意味着我们会详细地讨论服务的运维表现，将服务的性能与设计、配置或者实现结合起来讨论，对于如何解决问题提供建议。组织这样一个常规会议，随时将服务的性能与设计联系起来讨论，是一个非常强大的反馈渠道。

生产会议通常每周进行一次；鉴于SRE对于毫无意义的会议十分反感，这个频率似乎是合适的：有足够时间积累足够的素材使得会议有价值，同时又不会太频繁而总让人们找借口不参加。每次会议通常持续30到60分钟。如果会议过短，意味着某些东西没有充分讨论，或者意味着是服务内容太少。如果会议过长则意味着你可能陷入了细节讨论之中，又或者是待讨论的东西太多，应该按服务或者团队进一步拆分会议。

正如其他会议一样，生产会议应该有一个会议主席。很多SRE团队由不同的团队成员轮值主席，这样做使得每个人都有服务负责人的感觉。但是，确实不是每个人都有同等水平的主持技巧，但是由于轮值带来的集体参与感的价值很高，我们可以接受一些临时的次优性选择。更重要的是，这是提高个人主持能力的好机会。这也是SRE经常面临的事故协调情形下非常有用的一项技能。

在两个SRE团队视频会议时，如果一个团队比另外一个大很多，我们建议从较小的团队一边选择主席。更大的一方会自然而然地安静下来，一些不平衡的团队规模所造成的不良影响（这些会由于视频会议的延迟而变得更糟）将得到改善。[14]我们不知道这是否有任何科学依据，但它确实有效。

议程

组织生产会议有许多方法，正如SRE所负责的服务和采用的方法十分多样化一样。从某种程度上说，严格规定如何运行这些会议是不恰当的。然而，提供一个默认的议程（见附录F中的一个例子）可能像下面这样：

即将到来的生产环境变化

变更跟踪会议是整个行业内众所周知的一种做法，事实上有时候整个会议都是专门用来阻止变化实施的。然而，在Google生产环境中，我们通常会默认允许变更，这就需要跟踪变化的一些属性：开始时间，持续时间，预期效果等。这是该会议针对短期事务所提供的能见度。

性能指标

我们进行以服务为核心的讨论的主要方式之一就是谈论系统的核心指标，参见第4章。即使系统本周内没有显著的故障，讨论逐渐（或大幅度）的负载增加是很普遍的。通过追踪延迟、CPU利用率等数字随着时间推移的变化，可以建立起对一个系统性能的大概认知。

一些团队也会追踪资源的用量和效率，这也是一个有用的指标，它可以揭示那些速度较慢，但是也许更凶险的系统变化。

故障

这个部分会讨论那些值得写事后总结的问题，是一个不可或缺的学习机会。一个好的事后总结的分析，如在第15章中讨论的那样，应该总是可以引发很多讨论的。

紧急警报

这里讨论的是监控系统发出的紧急警报，涉及那些可能需要书写事后总结的问题。前述的故障环节关注层次更高的讨论，而这一部分则更着眼于战术层面：发生了哪些紧急警报，谁收到了紧急警报，后续又发生了什么事情等。这部分有两个隐含的问题：这个警报发出的时间、内容等是否合适？这个警报应该是紧急警报吗？如果对于后一个问题的答案是否定的，那么就应该去掉这个警报。

非紧急警报事件

这里包含三个项目：

- 一个应该有紧急警报发生却没有的问题。在这种情况下，你大概应该修复监控系统使得这样的事件会触发紧急警报。通常，这是在你试

图解决其他问题时发现的，或者是某个你一直跟踪的度量指标发生了变化，但是却没有对应的报警规则。

- 非紧急警报事件，但是却需要引起注意的。如影响范围较小的数据损坏问题，或者是非直面用户的系统部分的速度缓慢问题。在这里跟踪一些反应式的运维工作也很恰当。

- 非紧急警报事件，但是不需要引起注意的。这些警报应该被去掉，因为它们制造了额外的噪声，在值得注意的问题之外分散了工程师的注意力。

之前的待办事项

前面的详细讨论经常会引发一系列SRE需要采取的行动—修复这个和监控那个，开发一个子系统等。像其他任何会议一样追踪这些改进：将待办事项分配给具体的人，并跟踪进展情况。就算没有任何待办事项，也应该有一个明确的议程项目。持续性的交付是一个良好的信誉和信任的构建者。具体如何交付不重要，重要的是确定它已经确实交付了。

出席人员

SRE团队的所有成员都有义务出席。如果你的团队分散在多个国家和/或时区，这是特别必要的。因为这是作为一个团队互动的主要机会。

主要的产品负责人也应该参加这个会议，任何有合作关系的产品研发团队也应参加。一些SRE团队会将该会议分段，任何只针对SRE的问题都在前半部分出现；这种方式是可以的，如前所述，只要每个人在离开会议的时候都能有一个一致的概念即可。偶尔也会有其他SRE团队的代表出现，尤其是如果有一些大型的跨团队的问题需要讨论时。一般来说，该SRE团队，加上其他的主要团队都应该一起参加。如果由于关系原因不能够邀请你的产品开发伙伴来参加这个会议，你应该去修复这样的关系：可能采用的第一步是从那个团队中邀请一位代表来参加这个会议，或者寻找一个值得信赖的中间人来进行代理沟通，建立起一个健康

的交互模型。团队之间不能很好相处有很多原因，有很多书籍专门讨论如何解决这样的问题：这些信息对于SRE 团队同样非常适用。但是这里最重要的一点是要推动形成一个SRE与研发团队的反馈回路，不然SRE团队的很大一部分价值就丢失了。

有的时候，你会发现需要参加这个会议的团队，或者很多忙碌而重要的与会者需要你去逐个邀请。这里有一些技巧可以用来处理这些情况：

- 不那么活跃的服务可能只需要产品研发团队的一个代表出席，或者仅仅是获得产品研发团队的一个承诺就够了：承诺他们会阅读和评论会议日程。

- 如果研发团队非常大，选择一部分人作为代表。

- 那些忙碌却重要的与会者可以通过事先提供个人反馈和引导的方式参加，或者使用事先填写的议程方式参加（接下来会说明）。

我们讨论的大部分会议策略都是常识性的，再加入一点服务为导向的改变即可。让这个会议更高效和更具包容性的一个独特地方是使用Google Docs的实时协作功能。许多SRE团队都有这样一个任何工程师都可以访问的文档。维护这样一个文档可以使得两个实践变得可能：

- 用“草根”想法、评论和其他信息事先填充议程。

- 非常有效地进行会前多人并行议程准备。

我们充分利用了产品提供的多人协作功能。在文档中看会议主席打出一段字，接着看到别人在这之后加入了一个资源链接，然后看到另一个人整理了全句的拼写和语法是一种很神奇的感觉。这种协作能够更快地做完事情，同时也让更多的人觉得自己是团队中的一分子。

SRE的内部协作

显然，Google是一个跨国企业。由于SRE的应急反应机制和on-call的轮值机制，SRE非常适合成为一个分布式的、至少跨几个时区的组

织。这种分布的实际影响是我们对于“团队”有非常不固定的定义。这与一般的产品研发团队很不一样。我们有本地团队，有同地域的团队，也有跨大陆的团队，还有很多各种大小的虚拟团队等。这其实造成了一个责任、技能和机会很有意思的组合。这其中的许多方面应该会在任何足够大型的公司中存在（对于科技公司尤其明显）。鉴于大多数的本地协作并没有特别的障碍，更值得讨论的是跨团队、跨地域，以及虚拟团队的协作。

这种分布式的模式还和SRE的团队是如何组织的类似。SRE团队最主要的目标是通过掌握先进技术来创造价值，而先进技术的掌握往往是困难的，因此我们试图掌握相关系统或基础设施的一部分子集以降低认知的难度。专业化是实现这一目标的一种方法，比如，团队X只会为产品Y服务。专业化有很多优势，因为它能够大幅提高技术熟练度。但同时专业化也是有弊端的，因为它会导致局部化，忽视大局。我们要有一个清晰的团队章程来定义一个团队将要做什么—更重要的是不会做什么—但这并不总是那么容易。

团队构成

SRE团队的成员有着各种各样的技能，从系统工程到软件工程，以及组织能力和管理能力都有。我们唯一可以确定的是：成功的协作需要团队内充分的多样性。有很多证据表明，多样化强的团队在各方面都更强（参见文献[Nel14]），组建这样一个多样化的团队意味着需要特别注意沟通，防止认知偏差的出现，我们就不在这里详细介绍了。

正式地讲，SRE团队中有着“技术负责人”（TL）、“SRE经理”（SRM）和“项目经理”（也被称为PM、TPM、PGM）的角色。有的成员希望这些角色的责任被明确定义：因为他们据此可以迅速和安全地做出范围内的决策。另外一些成员则希望在一个更为动态的环境中工作，这样他们可以随时协商切换责任。普遍看来，越是动态的团队，团队成员的个人能力越强，整个团队适应新的情况的能力也越强。但是这样的团队需要在沟通上花费更多时间，因为什么东西都不太确定。

不管这些角色定义得是否清晰，基本来说，技术负责人是负责团队技术方向的，可以以多种方式进行领导。他可以通过仔细评审每个人的

代码，组织进行季度工作汇报，或者是在团队中推进共识的建立来引领团队方向等方式来领导团队。在Google内部，技术负责人和SRE经理的工作几乎一样，因为我们的SRE经理也具有高度的技术能力。但SRE经理在这之外还有两个特殊责任：绩效管理，以及其他一切其他人不能处理的事情。好的技术负责人、SRE经理以及项目经理组成了一个完整的管理团队，可以很好地组织项目进行讨论设计文档，必要的时候甚至亲自书写代码。

高效工作的技术

SRE采用很多方法提高工作效率。

一般来说，单人项目最终肯定会失败，除非此人个人能力超强或者待解决的问题是非常简单直接的。做成任何高价值的事情都需要很多人共同协作。正因为这样，SRE团队需要良好的协作能力。这里再提一下，有很多阅读材料都讨论了团队协作，这类资料的大部分也都适用于SRE。

一般来说，在做本地团队边界之外的工作时要想成功就一定需要良好的沟通技巧。而与异地团队合作，或者与跨越时区的团队一起工作则需要极好的书面沟通能力或者是大量的旅行，这样才能建立人与人之间的高质量关系。虽然有的时候这种旅行可以推迟进行，但是最终来看仍是必要的。书面沟通能力再强，随着时间的推移，人们对你的印象也会逐渐淡薄成一个电子邮件地址，直到你再次出现进行直接沟通。

SRE内部的协作案例分析：Viceroy

一个成功的SRE内部跨团队协作案例叫作Viceroy，该项目是一个监控台（dashboard）框架，也是一项服务。目前的SRE的组织架构经常会导致不同团队产出多种目的相同、实现相似的项目；在多种因素作用下，监控台程序在这个问题上尤其严重。[15]

造成团队产生许多废弃的、遗留的监控框架的原因是非常容易理解的：每个团队都可以通过开发自己的解决方案获得迅速的直接回报，跨越团队边界的工作通常是很艰难的，同时Google基础设施通常提供的是

一个SDK而非完整的一套产品。这些环境因素其实就是在鼓励每个工程师使用该SDK创造出另外一个废弃品，而不是尽可能地为更多人同时解决一个问题。

Viceroy的诞生

Viceroy 是与众不同的。该项目开始于2012年，当时许多SRE团队正在考虑如何迁移到Monarch—Google内部新的监控系统上。SRE对于监控系统的态度是非常保守的，所以SRE团队比非SRE团队花了更多时间来接受Monarch（这很讽刺，因为Monarch的设计目的就是为了方便SRE团队）。但是，大家都知道过时的监控系统 Borgmon（见第10章）有很多可以改进的地方。例如，该系统的监控台就很难用，因为它用的是自己开发的一套HTML模板系统，这个系统不符合任何标准，充满了各种边界条件，非常难以测试。当时，Monarch已经足够成熟，原则上获得了SRE团队的一致认可即可成为Borgmon的替代品，在内部也有越来越多的团队采用。但是在替换过程中，我们发现新系统在监控台功能上仍然存在问题。

那些尝试使用Monarch的人很快发现它在监控台功能上有两个不足：

- 为小型服务建立监控台页面很容易，但是它对于一个拥有复杂监控台页面要求的服务没有很好的扩展能力。

- Monarch不支持过去的监控系统模板，这样使得迁移到Monarch非常困难。

由于当时没有一套可行的方案来部署Monarch，好几个团队开始各自闭门造车。由于当时SRE管理层没有足够的跨团队项目跟踪和协调能力（现在这一问题已被解决），我们又一次造成了很多重复项目的诞生。在12～18个月的时间内，很多团队—包括Spanner团队、广告前端服务团队以及一些其他服务团队都分别开始了自己在这个方面的工作（这里一个值得提到的例子是Console＋＋项目）。最终理智占据了上风，各个团队之间终于发现了彼此之间重复性的劳动。这些团队最终决定一起做更明智的事情—联合起来创造一个对于所有SRE团队通用的解

决方案。由此，Viceroy项目诞生于2012年年中。

自2013年年初，那些没有迁移到新系统，但是很想尝试的团队开始对Viceroy产生了兴趣。很显然，团队现存的监控系统越庞大，他们切换的动力越小：现有系统的维护压力很小，替换成一个未经验证的新系统又很麻烦。监控系统本身的需求就很多样化，加上团队又很不情愿在上面花时间，动力就更小了。但是，所有的监控台系统都有两个主要功能需求：

- 支持复杂的、自定义的监控台页面。
- 同时支持Monarch和Borgmon。

同时每个项目也有它们自己的技术要求，这些依赖于作者的偏好或者经验。比如说：

- 核心监控系统之外的多样化数据源。
- 用配置文件自动生成的页面或者是用明确的HTML布局的页面。
- 没有JavaScript的页面和大量使用Javascript与Ajax的页面。
- 使用全静态内容，这样页面可以缓存在浏览器中。

由于这么多复杂要求的存在，全面融合到一个框架里非常困难。事实上，虽然Console＋＋团队很乐意与Viceroy整合，他们在2013年上半年的实验证明了两个项目之间的根本差异造成整合基本不可能。这其中的最大困难是，Viceroy的设计中就没有采用过多的JavaScript，而Console＋＋则是以JavaScript为主。但是，我们仍有一线希望，因为这两个系统中确实有一些基本的相似性：

- 它们的HTML模板进行渲染的语法很类似。
- 它们的一些长期目标是一致的，这两个团队也都还没有开始解决。例如，这两个系统都希望缓存监控数据并支持离线定期产生那些实时计算过于耗时的控制台数据。

最终，我们暂时搁置了对于统一监控台框架的讨论。然而，到2013年年底，Console＋＋和Viceroy都有了显著的发展。它们之间的技术差距已经缩小，因为Viceroy也开始使用JavaScript来呈现图表。两个团队再次碰头，发现整合最终需要的就是用Viceroy服务器来提供Console＋＋数据。第一个集成的原型在2014年年初完成，证明了两个系统可以很好地一起工作。这时，两个团队都愿意在整合上继续投入力量，因为Viceroy当时已经成为一个人人皆知的监控解决方案的品牌，联合项目就延续了Viceroy这个名字。开发其他的功能又花费了几个季度的时间，2014年年底，整合系统正式完成。

整合过程带来了巨大的收益：

- Viceroy 增加了大量的数据源支持，同时增加了一个JavaScript客户端。

- JavaScript编译过程被重写，以支持独立的模块。这些模块可以被选择性地包括在其中。这对使用自己的JavaScript代码的团队来说是必需的。

- Console＋＋项目从Viceroy的很多改进中也获益不少，例如缓存和背景数据渠道的增加。

- 总的来说，在整合项目上的开发速度比其他各重复项目速度的总和大得多。

最终看来，对未来的一个共同愿景是项目合并成功的关键因素。这两个团队在开发过程中都发现了自己的价值，并且从对方的贡献中受益。这种势头一直保持到2014年年底，Viceroy被正式宣布为适用于所有SRE团队的通用监控解决方案。这个声明带有Google的一贯特色，它没有“要求”团队采用Viceroy：相反，它“建议”团队采用Viceroy而非另行开发监控台程序。

所面临的挑战

虽然Viceroy最终取得了成功，但过程中遇到了很多困难。其中的许多困难都是由于跨地域的团队沟通所造成的。

在新的Viceroy团队建立之初，远程团队成员之间的协调是非常困难的。因为人与人之间的沟通方式差异很大，第一次见面时书面表达习惯和口语表达习惯中隐含的微妙暗示很容易被误解。在项目开始之初，那些不在总部工作的团队成员经常会错过会议开始之前和结束之后立刻进行的即兴讨论（现在的沟通渠道已经大大改善了）。

虽然Viceroy的核心团队一直保持一致，由其他贡献者组成的扩展团队却变化很快。随着时间推移，贡献者的职责也发生了变化，以及因此许多人只能够坚持参与项目1～3个月的时间。因此，比Viceroy核心团队规模大得多的贡献者群体有很多的人员变动。

在项目中增加新人需要对其进行系统总体设计和结构的培训，这需要一定的时间。然而，当某个SRE贡献了Viceroy的某个功能之后回到自己的团队时，他们就成为Viceroy的本地专家，这种本地专家带来了更多的用户。

随着人们不停地加入和离开团队，我们发现，这种短期的贡献是有用的但也是昂贵的。其主要成本是所有权的稀释问题：一旦某个功能做完了，维护者离开团队之后，这个功能会随着时间的推移出现各种问题，最终一般会被丢弃。

此外，Viceroy项目的范围也随着时间的推移而增长。该系统在上线时有着雄心勃勃的目标，但最初的项目范围还是有限的。然而随着项目范围的不断扩大，核心功能的按时交付变得越来越困难，同时还要改善项目管理，设置更清晰的方向以确保项目一直走在正轨上。

最后，Viceroy团队发现很难完全掌握一个由远程贡献者维护的组件。即使大家的出发点都是好的，人们总是会默认选择阻力最小的路径，在不涉及远程交流的情况下讨论问题或做出决定，这可能会导致一定的冲突发生。

建议

只有在不得已的情况下才应该跨地域开发项目，但是这同时也会带来一定的好处。跨地域工作需要更多的沟通，工作完成得也更慢；但是

好处是—如果你能协调好的话—则会有更高的产能。单地项目其实也可能会导致其他人不知道你正在做什么，所以这两种做法其实都有一定成本。

动力十足的贡献者是有价值的，但不是所有的贡献成果都是同样宝贵的。确保项目的贡献者会实际投入时间，而不只是为了一些朦胧的自我实现的目标而加入（例如想要在一个闪亮的项目上带上自己的名字；想要在一个新的令人兴奋的项目上编码，却不承诺未来的维护）。有着特定目标的贡献者通常会更有动力，能够更好地维护他们的贡献。

随着项目的发展，项目规模也会扩大，肯定会有其他团队的成员参与进来。因此，我们应该仔细考虑一下项目的框架。项目负责人非常重要：他们为项目提供一个长期愿景，确保所有的工作都与这个愿景相关，设置正确的工作优先级。同时，我们还需要一个普遍认同的做决策的方式。如果彼此有高度的认同感和信任感的话，应该倾向于多做一些本地决策。

标准的“分而治之”的策略适用于跨地域项目；通过将项目拆分成尽可能多的合理大小的部分可以减少沟通成本。努力确保每一个组件都可以被分配给一个小组，最好是保持在同一个地域范围内。如果将这些组件分散给项目的子团队，应建立明确的目标和最后期限。（同时还要试着不要让康威定律太多地影响软件的自然状态。）[16]

只有当团队的目标是提供某个功能或解决某个问题时，才是最高效的。这样确保了团队中的每个人都知道整个团队对他们的期望，并且知道只有该组件完全集成并使用在主项目中，他们的工作才算完成。

显然，一般的软件工程最佳实践也适用于协作项目：每一个组件都应该有设计文档并应该在团队内部评审。这样，团队中的每个人都有机会及时了解变化，也有机会参与影响和改进设计。文档化是抵消物理和/或逻辑距离的主要技术之一，一定要多用。

标准化是很重要的。编码风格指南是一个很好的开始，但它们通常是战术性的，只是一个建立团队规范的起点。每当发生一次辩论时，都应该在团队内部进行充分地论证，但是也要设置一个严格的时间限制。

选择一个解决方案，将它记录下来，然后继续推进项目。如果不能够达成一致，就需要挑选一个每个人都尊重的仲裁员来做出决定，并且再次向前推进。随着时间的推移就能建立一个最佳实践的集合，这会有助于新人更快上手。

虽然一些当面交流可以用视频会议和良好的书面沟通暂时替代，最终的当面交流还是必要的。如果可以的话，应该让项目的领导者亲自与其他成员会面。如果时间和预算允许的话，应该组织一个团队会议，让团队的所有成员都可以进行互动。举办一次峰会是一个解决设计问题和目标的好机会。在中立性很重要的情况下，应该选择在一个中立的位置举办峰会，这样可以让与会者都没有“主场优势”。

最后，采用一个适合当前项目状态的项目管理方式。不管多么雄心壮志的项目也要从小规模开始，所以项目管理成本应该相应保持较低的水平。随着项目的发展逐渐改变项目的管理方式。如果项目增长迅速，那么也有必要开展更全面的项目管理。

SRE与其他部门之间的协作

正如前文所述，以及第32章中描写的那样，产品开发组织和SRE之间的合作最好发生在设计阶段的早期，最理想的状态是在任何一行代码还没有被提交之前。SRE最适合为架构和软件的行为提供建议，后期进行这些改造是相当困难的（如果可能的话）。在设计新系统时，有SRE参与对所有人都有帮助。一般来说，我们使用目标和关键结果（OKR）流程（参见文献[kla12]）来跟踪这样的工作。对于某些服务团队，这样的协作是他们主要的工作—跟踪新的设计，提出建议，帮助实施这些建议，并且一直部署到生产环境。

案例分析：将DFP迁移到F1

从现有的服务迁移到新的服务上在Google内部是很常见的。典型的例子包括将服务组件移植到一个更新的技术上，或者是更新组件以支持新的数据格式。随着全球部署的新数据库技术的引入，如Spanner（参见文献[Cor12]）和F1（参见文献[Shu13]）,Google内部进行了多个涉及数据库的大型迁移项目。其中一个项目是将DoubleClick for

Publishers（DFP）[17]的主要数据库从MySQL迁移到F1。值得注意的是，本章的一些作者就负责该服务系统的一部分。如图31-1所示，该系统不断地提取和处理来自数据库的数据，生成一系列索引文件，以便在全球各地对外服务。该系统分布在多个数据中心中，并且使用约1000个CPU和8TB的内存来索引每天100TB的数据。

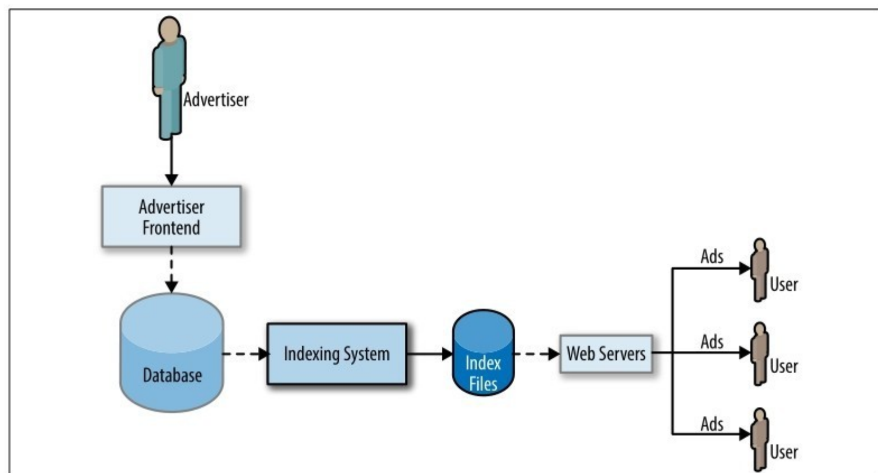


图31-1：一种通用的广告服务系统

迁移过程很复杂：除了迁移到一个新的技术之外，由于F1提供了在列中存储和索引Protobuf的能力，数据库表结构也被大幅重构和简化过了。目标是将数据处理系统迁移到F1上，让它可以产生一个与现有系统相同的输出。这可以使我们不修改任何服务相关的组件，这样从用户的角度来看是一次无缝迁移。另外一个附加的限制条件是，该产品需要我们完成一个实时的迁移，而且不能中断服务。为了实现这一目标，产品研发团队和SRE团队从一开始就密切合作开发新的索引服务。

作为该系统的主要开发人员，产品研发团队通常来说更加熟悉软件的业务逻辑（business logic），并且他们与产品经理和实际产品的“业务需求”联系更紧密。相反，SRE团队通常对于软件的基础设施建设部分有更多的专业知识（例如分布式存储系统或数据库的客户端类库），由于SRE经常在不同的服务之间重用相同的技术，他们有很多保证软件可扩展性和可靠性方面的知识。

从该项目一开始，产品开发和SRE团队就都清楚他们必须建立起非

常紧密的合作，最后通过举行每周例会的方式来同步项目的进度。在这个具体案例中，业务逻辑的变化在一定程度上依赖于基础设施的变化。由于这个原因，项目是以设计新的基础设施开始的；拥有可扩展地进行提取和处理数据的广泛知识的SRE们主导了对基础设施改动的设计过程。这涉及设计如何从F1中提取出不同的表，如何过滤和合并数据，如何只提取改变了的数据（而不是整个数据库），如何承受一些物理机器的损失而不影响服务，如何确保资源使用量与提取数据量维持线性增长，容量规划，以及其他许多方面。新的基础设施方案与其他现存的从F1提取和处理数据的服务很类似。因此，我们可以保障这个方案的可行性，同时也可以重用现存的部分监控组件和工具。

在新的基础设施进行具体开发之前，两个SRE书写了一份详细的设计文档，产品研发团队和SRE团队都对该文档进行了详细的评审，调整了一些细节以应对一些边缘情况，最终敲定了一个全体同意的设计方案。这样的一个文档清晰地定义了新的基础设施的变化，业务逻辑可以将此作为依据进行设计。比如，设计中新的基础设施仅仅取出变化的数据，而不是重复地取出全部数据；业务逻辑层面就必须适应这个新的做法。在很早之前，我们就明确定义了基础设施和业务逻辑之间的接口，这样可以让产品研发团队独立进行业务逻辑的开发。同样的，产品开发团队会让SRE随时了解业务逻辑的变化。当业务逻辑与基础设施互相依赖的时候（例如业务逻辑的变化依赖于新的基础设施时），这种协调方式可以让两个团队更快了解变化，并且更快速地实施变化。

在该项目的后期，SRE开始在一个与生产环境很接近的测试环境中部署新的服务。这个步骤对确认新服务的正确性是非常有必要的—特别是在性能和资源利用率方面—这时业务逻辑的开发仍在进行中。产品开发团队使用这个测试环境来进行新服务的正确性测试：旧的服务（在生产环境中运行的）产出的广告索引必须与新服务（在测试环境下运行）产出的索引相匹配。不出所料，验证过程中显示出了新老业务逻辑之间存在的差异（由新的数据格式下的一些边缘情况导致）。产品开发团队能够利用这个环境快速迭代解决这些问题：针对每一个广告他们会找出造成差异的原因，以便修复产生问题的业务逻辑。与此同时，SRE团队开始准备生产环境：在不同的数据中心配置必要的资源，运行程序和增

加监控规则，同时对on-call工程师进行培训。SRE团队同时建立了一个基本的发布流程，其中包括对发布结果的校验。这种任务通常由产品研发团队或是发布工程师完成，但在这个案例中是由SRE完成的，因为这样可以加快迁移速度。

当服务准备好之后，SRE与产品开发团队共同准备了一份发布计划，SRE进行了最后的发布。发布过程非常成功，非常平稳，没有造成任何用户可见的问题。

小结

鉴于SRE团队全球分布的特性，有效的沟通是非常重要的。本章讨论了SRE团队用来维持SRE团队之间以及和各种伙伴团队有效联系的工具和技术。

SRE团队之间的协作存在一定的困难，但是也存在巨大回报。这种回报包括建立一套解决问题的通用方法，以及可以集合更多力量解决更困难的问题。

第32章 SRE参与模式的演进历程

作者：**Acacio Cruz**、**Ashish Bhambhani**

编辑：**Betsy Beyer**、**Tim Harvey**

SRE 参与模式：是什么、怎么样以及为什么

本书余下的大部分内容讨论的是当SRE已经在负责运维某项服务的工作时会发生什么。Google内部只有很少一部分的服务是在一开始就由SRE负责运维的，这就意味着SRE需要有一个评估服务的流程，该流程确保服务符合SRE的要求，同时与研发团队协商如何改善任何不足之处，以及确定最终交接给SRE运维的流程。我们将该流程称为接手（onboarding）。如果你目前的环境中有许多完善程度参差不齐的服务，那么新组建的SRE团队一般按照优先级顺序逐步接手各种服务的运

维工作，低优先级的服务要等待高优先级的服务接手之后才会被考虑。

虽然这种做法是很常见的，而且是处理既成事实的合理的方式，但是我们在这里提出，至少有两种更好的方式可让SRE用他们的丰富生产环境经验帮助新老服务。

首先，和软件工程学中的经验一样—错误越早被发现，就越容易修复它—SRE参与的时机越早，该服务的质量提升越快，最终质量也越好。当SRE从早期设计阶段就介入时，新服务“接手”阶段会很短，而且一般“出厂”时就非常可靠。通常这是因为我们不需要再花时间去修复设计或者实现中隐藏的问题。

其次，也许是最好的方式，与其创造很多各异的个体系统交给SRE运维，不如直接让研发团队在一个通过SRE验证的基础设施平台上进行产品开发。建立这个平台将具有双重效益，让产品变得更可靠，同时又使整体流程更具扩展性。这能够避免一定的认知过载问题，并且通过推广公共基础设施的做法，让产品开发团队专注于他们最应该关注的地方——在应用程序层的创新。

在接下来的章节中，我们会讨论上述每一种参与模型。先从最经典的PRR驱动模型开始。

PRR 模型

SRE参与一个项目时，最典型的第一步是进行生产就绪程度评审（PRR）。该流程根据服务的具体细节来找出在可靠性方面的欠缺之处。通过PRR流程，SRE寻求运用他们所学到的经验来确保服务生产运行中的可靠性。只有通过PRR评审，SRE团队才会同意承担该服务的运维责任。

图32-1描绘了一个典型服务的生命周期。PRR评审可以在其生命周期的任何一点开始，但是随着时间的推移，SRE会参与越来越多的阶段。本章介绍了一种简单PRR模型，然后讨论SRE是如何通过对这种模型的修改，也就是扩展参与模型（Extended Engagement Model）以及SRE平台的构建来扩大SRE的影响力的。

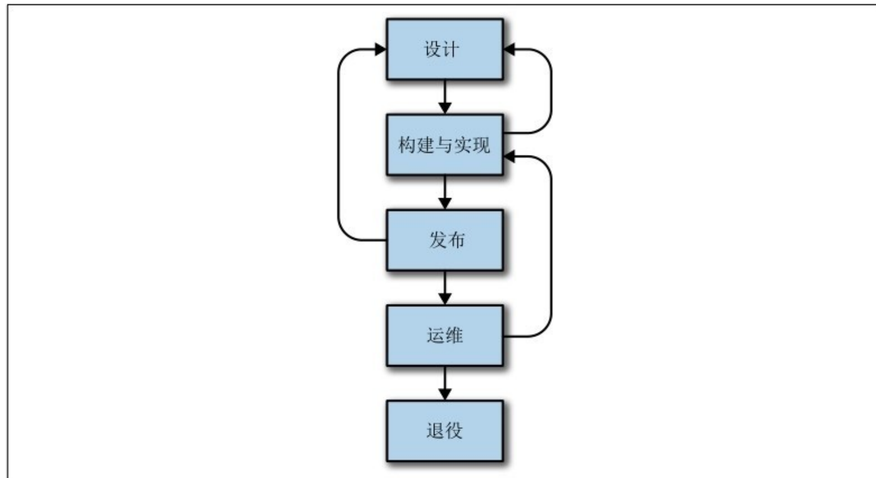


图32-1：一个典型的服务生命周期

SRE参与模型

SRE承担重要服务的生产运维的责任，是为了提高该服务的可靠性。SRE会考量该服务的几个方面，统称为“生产方面”。这些方面包括以下内容：

- 系统的体系结构和跨服务依赖
- 指标的选择、度量和监控
- 紧急事件处理
- 容量规划
- 变更管理
- 性能：可用性、延迟和资源效率

SRE参与运维服务的目标是在这些方面进行改善，这样可以让该服务的生产运维变得更容易。

替代性支持

不是所有的Google服务都有SRE的紧密参与。有以下几个原因：

- 许多服务不需要高可靠性和高可用性，可以通过其他方式支持该服务。

- 开发团队的数量总是会超过SRE团队的可支持的数量（参见第1章）。

当SRE无法对某项服务提供全方位的支持时，我们会提供其他选项，例如文档和咨询工作来帮助研发部门改进其生产运维。

文档

SRE对内部的技术与那些被广泛使用的系统提供了开发指南。Google的生产指南（production guide）记录了生产运维中的最佳实践，由那些经验丰富的SRE团队以及研发团队负责维护。开发者可以通过遵循这些建议，实现其中描述的方案来提高他们的服务质量。

咨询工作

开发者同时也可以寻求SRE咨询，来讨论服务的具体问题。发布协调工程团队（LCE）（参见第27章）的大部分时间都是与开发团队进行咨询工作。其他的SRE团队也经常参与这些咨询。

在一项新的服务或新的功能开发完成之后，开发者在进入发布阶段之前通常会征求SRE的意见和建议。这种发布前的咨询工作通常需要一个或两个SRE花费几个小时来学习设计和实现要点。负责该咨询工作的SRE随后会与研发团队碰面，指出一些风险性较大的地方，并且讨论可以用于提高生产服务质量的一些常见解决方案。这里的建议可能直接来自于前面提到的“生产指南”。

这种咨询工作的范围肯定是非常宽泛的，因为一般来说，SRE也不可能在有限的时间内对于一个给定的系统有非常深刻的理解。所以，对于某些研发团队而言，这种咨询是不够的：

- 某个服务自发布到现在，已经进行了数个数量级的增长。该服务现在已经不能通过仅仅阅读文档以及简单的咨询来解决问题，必须投入更多的时间来仔细了解。

- 某个服务发布之后，逐渐被许多其他的服务所依赖。流量有明显增长，同时这些流量来自许多不同的客户。

上述这些类型的服务可能已经发展到了一个转折点，这时它们会开始在运维过程遭遇重大困难，同时这些服务对用户来说又越来越重要。因此，为了确保这些服务不断发展的同时可以被合理运维，有必要进行长期性的SRE参与。

PRR：简单PRR模型

当SRE接到来自研发团队的生产运维接管要求时，他们需要评估该服务的重要程度和目前团队的可用性。如果该服务的确值得SRE支持，并且SRE团队和研发团队同意调动人力资源来促成这一支持，那么SRE会与研发团队一起启动PRR评审流程。PRR评审的目标如下：

- 验证该服务符合公认的标准的生产部署方式和运维方式，同时该服务负责人已经准备好与SRE一起工作，利用SRE的知识改进服务。

- 提升服务生产环境中的可靠性，并最小化可预见的故障的数量和严重程度。PRR会关注SRE所关注的生产环境的所有方面。

在必要的改进全部实施之后，SRE认为该服务已经“准备好”之后，一个SRE团队会接管所有的生产运维职责。

接下来，我们会讨论PRR评审流程本身。一共有三个不同但是相关的参与模型，它们是简单PRR模型，早期参与模型，框架和SRE平台模型，我们接下来将依次对它们进行讨论。

我们将首先描述简单PRR模型，这通常是针对已经推出并且即将由SRE团队接管的服务。PRR流程包含以下几个阶段，和研发生命周期很像，但是这里它可以与研发周期并行进行。

参与

SRE管理层首先需要决定具体由哪个SRE团队接管服务最合适。通常管理层会选择1～3名SRE，或者是由SRE自我提名来负责PRR过程。

该小组负责与研发团队启动讨论。该讨论涵盖以下事项：

- 为该服务建立一个SLO/SLA。
- 为可能的、为了提高可靠性的破坏性设计变更制定计划。
- 制定交接计划和培训计划。

该流程的目标是对PPR流程、最终目标，以及结果形成一个共识。

分析

工作中的第一个较大阶段是分析。在这个阶段中，SRE评审者将仔细学习该服务的一切知识，开始对其生产方面的缺陷进行分析。SRE的目标是度量该服务自身的成熟程度，以及SRE最关注的运维方面的情况。他们还会审阅服务的设计文档以及具体实现，以确保遵循生产环境的最佳实践。通常，SRE会在这个阶段专门建立以及维护一个PRR检查列表。该检查列表是针对该服务特别制定的，一般是基于领域专业知识，以及相关或类似系统的经验，加上“生产指南”中的最佳实践得出的。SRE团队同时也可以向对某些组件更有经验，或者是对被依赖服务有运维经验的其他SRE团队进行咨询。

检查列表中的项目可能包括：

- 对于服务进行的更新是否立刻影响到系统中大比例的部分？是否合理？
- 服务的依赖是否合理？例如，直接面向用户请求的服务不应该依赖于一个主要用于批处理的系统。
- 该服务在连接关键的远程服务时，是否需要较高的网络服务质量（QoS）？
- 该服务是否会将错误报告传送到中央日志记录系统进行分析？该服务是否报告所有的会造成降级回复，或者会造成最终用户请求失败的特殊情况？

- 所有用户可见的请求失败情况都有度量和监控吗？报警策略是否合适？

该检查列表可能还会包括该SRE团队所遵循的运维标准和最佳实践。例如，一个功能完善的但是没有遵循该SRE团队的“黄金标准”的服务配置，可能需要重构，以便更好地与SRE工具结合。SRE同时也会查看最近事故与对应的事后总结，包括事后总结中列出的待办事项。这个评估过程可以评估该服务对紧急事件响应的需求程度，以及其是否具有良好的运维操控性。

改进和重构

上文的分析阶段会形成一系列对于该服务改进的建议。下一阶段按如下顺序进行：

- 将每个改进建议按照其对服务可靠度的影响而排序，制定一个优先级顺序。
- 与研发团队讨论和协商这些建议，达成一个共识。
- SRE和产品研发团队同时参与，并且协助对方进行重构，或者实现额外的功能。此阶段所需要的时间和资源在各个项目上有很大不同。这主要取决于工程师花在重构上的时间，评估该服务成熟度和复杂度的时间，以及无数的其他因素。

培训

服务生产运维的责任通常是由整个SRE团队共同承担的。为了确保整个团队都有所准备，领导PRR的SRE评审者负责对团队的培训，这还包括整理运维服务所必需的文档。一般来说，通过研发团队的帮助和参与，这些SRE会组织一系列的培训课程和练习。其中包括：

- 设计概述
- 对系统中各种请求的处理流进行深入探讨
- 生产环境部署模式的描述

- 对系统运维各个方面的手动练习

当培训结束时，SRE团队应该已经做好接管服务的准备了。

“接手”服务

培训阶段之后，SRE团队进入具体的“接手”阶段。该阶段涉及将各种生产运维职责和所有权逐步转移给SRE，其中包括部分的运维操作、变更管理流程以及访问权限控制等。SRE团队将继续专注于之前提到的生产的各个领域。为了保障交接彻底完成，研发团队必须在这段时间内给予SRE团队后备支援与各种建议。这种关系将成为团队之间未来工作的基础。

持续改进

活跃的服务会根据新的需求和条件持续变化，包括用户请求的新功能，不断变化的系统依赖，技术升级和其他各种因素。SRE团队必须在面对这种持续改动的同时维持服务的可靠性标准。随着平时的生产运维，新变更的评审，紧急事件的处理，特别是书写事后总结，以及对于根源问题的分析，该SRE团队自然而然地会对服务了解更加深入。这种专业知识会分享给研发团队，形成对新功能、新组件和新的依赖关系加入时的建议和意见。这些运维经验同时会形成一系列的最佳实践，这些会记录在“生产指南”和其他文档中。

参与“莎士比亚”项目

一开始，莎士比亚搜索服务的研发人员对产品直接负责，包括对紧急事件进行响应。然而，随着服务用量的上升和服务收入的增长，该服务开始需要SRE的支持。由于该产品已经上线，所以SRE启动了PRR评审流程。评审中发现的一个问题是该服务的监控台页面漏掉了一些SLO指标，需要修复。在全部问题都被修复以后，SRE接过了该服务的on-call职责，但是同时还有两个开发者参与运维轮值。开发者同时也会参与每周一次的on-call会议，讨论上周遇到的问题以及如何处理即将到来的大规模维护活动或者集群离线情况。同时，有关服务的未来计划也会与SRE充分讨论，这样确保未来新的发布更平稳（然而墨菲定律总是寻找

机会搞破坏）。

简单PRR模型的演进：早期参与模型

到目前为止，我们刚刚讨论了PRR流程在简单PRR模型中的应用，这仅适用于已经进入发布阶段的服务。该模型同时还有很多限制和对应的成本。例如：

- 团队之间的额外沟通会增加开发团队的开销，同时对SRE评审者也造成了一定的认知负担。

- 合适的SRE评审者很关键，他们必须有时间参与，并且能够在现有事务之外推进PRR流程。

- SRE所做的工作必须是高度可见的，同时能够被研发团队充分评审，以确保知识的共享。SRE实际上应该作为开发团队的一部分工作，而不是作为外部单元参与。

然而，PRR模型的主要限制来源于这样一个事实：服务已经发布，已经规模化，SRE的参与是在生命周期的后期才进行的。如果PRR发生在服务生命周期的早期，SRE修复服务中潜在问题的机会将显著增加。这样一来，SRE参与的成功率以及服务自身未来的成功率都有可能提高。

早期参与模型的适用对象

早期参与模型通过在开发周期的早期引入SRE来实现显著的额外优势。应用早期参与模型需要在研发周期的早期就确定服务的重要性和/或商业价值。还需要确定该服务是否将来会有足够的规模或复杂性，以从SRE的专业知识中受益。适用的服务往往具有以下特点：

- 该服务实现了一个显著的新功能，并将成为一个现存的SRE运维的系统的一部分。

- 该服务是一个针对现有系统的大幅重写或者替代品，服务目标一致。

- 研发团队曾经寻求过SRE的建议，或在发布之后就需要SRE接管。

早期参与模型本质上就是将SRE尽早投入到研发过程中。SRE关注的焦点仍然不变，只是改善生产质量的手段不同。SRE会参与设计和后面的各阶段，最终在“构建”阶段或者其后的任何一个阶段接管。该模型是基于研发团队和SRE团队之间的积极合作建立的。

早期参与模型的优势

虽然早期参与模式的确带来了一定的风险和困难，但这种将SRE专家的经验和协作应用于整个服务生命周期，而非后期的方式也产生了很大的益处。

设计阶段

在设计阶段与SRE的合作可以预防未来很多类型的问题或事故的发生。虽然在开发生命周期中，设计决策可以被逆转或纠正，但这样的变更所需要的成本和复杂性都很高。最好的生产事故是那些从来没有发生过的事故！

偶尔，艰难的权衡取舍导致研发团队选择了一个不太理想的设计。通过在设计阶段的参与，SRE可以从一开始就了解这种权衡取舍，同时成为决策的一部分。早期SRE介入的目标是最小化服务进入生产后出现的设计分歧。

构建和实现

构建阶段涉及生产中的很多方面，例如指标选择与度量，运维和紧急事件中的控制功能，资源的使用率和效率等。在这个阶段，SRE可以通过建议使用现存的类库和组件来影响和优化具体实现，或者直接帮助构建某些控制系统。SRE在这个阶段的参与有助于使未来运维操作更便利，同时允许SRE在发布之前就获得运维经验。

发布

SRE还可以帮助实现广泛使用的发布模式和控制系统。例如，SRE可以帮助实现一种“隐形发布”机制，将现有用户流量的一部分复制发送

给新服务，不影响生产服务。新服务中的响应是“隐形”的，因为它们会被直接抛弃而不会实际显示给用户。这种“隐形发布”的做法可以让团队获得运维知识，在不影响现有用户的情况下解决问题，并减少在发布后遇到问题的风险。发布过程中的平稳有助于保持运维负载很低，同时可以维持在发布之后的发展势头。发布过程中出现的问题会很容易地导致对于源代码和生产环境的紧急更改，这会打乱研发团队对于未来功能的计划。

发布之后

发布时维持系统稳定一般来说可以减少研发团队的优先级冲突问题。他们不会在提高服务的可靠性与增加新功能之间进行抉择。在服务的后期阶段，这些早期经验可以帮助更好地进行重构或重新设计。

通过参与范围的扩大，SRE团队可能会比在简单的PRR模型下更快地接管新服务。在这种模型中，SRE和开发团队之间的更长时间和更密切的协作会创建一个长期协作关系。积极的团队关系会促进团结互助，帮助SRE尽早接管生产环境。

从服务脱离

有时候一个服务最终不值得SRE全职管理—这可能是在发布之后决定的，或者SRE可能曾经参与服务但从未正式接管。这其实也是好的，因为该服务已经足够可靠，维护成本很低，因此可以由研发团队运维。

也有可能是，SRE进行了早期参与，但是最后该服务不能满足预测的使用水平。在这种情况下，SRE所花费的精力是该新项目带来的整体业务风险的一部分，并且与达到预期规模的项目成功对比之下的较小部分。SRE团队可以被重新指派，之前所吸取的经验教训可以纳入新的参与过程中。

不断发展的服务：框架和SRE平台

早期参与模型超越了简单PRR模型。然而，SRE参与模型还有可以演进的空间，主要在于对可靠性的设计方面。

经验教训

随着时间的推移，目前描述的SRE参与模型产生了以下几种不同的情景：

●“接手”每个服务需要2～3个SRE，该过程通常持续两到三个季度。PRR模式的前期时间通常很长（多个季度）。所需花费的资源与服务的数量成正比，同时还受到可以主持PRR的 SRE数量的限制。这些问题导致服务交接是顺序进行的，而且要求制定严格的服务优先级。

● 由于各个服务中软件实践的不同，每个生产功能的实现方式都不同。为了达到PRR标准，各种新功能通常必须专门为每个服务重新实现，或者在最好的情况下，也要为所有共享代码服务子集实现一次。这些重新实现是对工程师工作量的浪费。一个典型的例子是，在同一种语言中，因为不同的服务没有实现相同的编码结构，功能相似的日志框架被反复实现。

● 通过对常见的服务问题和故障的评审揭示了一定的模式，但是并没有办法可以轻松地在其他服务中复制这些修复和改进。典型的例子包括服务过载的处理和数据热点的处理。

● SRE软件工程对于服务的贡献往往是本地的。因此，建立能够重复使用的通用解决方案是困难的。这样的后果是，没有一个简单的方法来将SRE团队收获的新经验在那些已经接手的服务中实践。

影响SRE的外部因素

外部因素传统上会对于SRE组织和它多方面的资源造成影响。

跟随行业趋势，Google内部也在逐步向微服务转变。[18]于是，需要SRE提供支持的数目和SRE需要负责的服务数量都增加了。因为每项服务都有一个基础固定的运维成本，就算是很简单的服务也需要更多的人手。微服务同时意味着部署时间应该较短，这是以前的PRR模型所不可能满足的（它有几个月的提前期）。

招聘合格的SRE很困难，代价也很高。尽管招聘组织为此付出了巨

大努力，我们仍然一直没有足够的SRE来支持全部服务。SRE入职之后，他们的培训时间相比一般的研发工程师也更长。

最后，SRE这个组织有义务满足数量庞大并且不断增长的目前没有SRE支持的开发团队的需求。这个义务促使SRE支持模型的扩展，使其远远超出了原有的概念和模式。

结构化的解决方案：框架

为了有效应对这些情况，很有必要开发一个能符合以下指导思想的模型：

最佳实践代码化

将生产环境中运行良好的最佳实践代码化，这样服务可以通过简单地使用这些代码，自然而然地成为“生产就绪”。

可重复使用的解决方案

常见并易于共享的技术实现，用于改善可扩展性和可靠性的问题。

带有通用控制界面的通用生产操作平台

生产设施的统一接口，统一的运维控制机制，以及统一的监控、日志以及服务配置。

更简易的自动化和更智能的系统

通用的控制接口使自动化和智能化达到一个以前不可能达到的水平。例如，SRE可以用一个统一的视图查看关于一次故障的全部相关信息，不用收集和分析来自不同数据源的原始数据（日志、监控数据，等等）。

基于这些指导思想，在我们支持的每一个环境中（Java、C＋＋、Go），建立了一系列SRE支持的平台和服务框架。利用这些框架构建的服务可以共享那些按SRE支持平台设计的代码，SRE和开发团队共同维护这些框架。这种变革使得产品研发团队可以利用符合SRE标准的框架

来设计应用程序，然后再花时间按SRE规范改造程序。

应用程序由一些业务逻辑组成，而这些业务逻辑依赖于各种基础设施组件。SRE所关注的生产问题主要集中在一项服务的基础设施相关的部分。服务框架以一个标准化的方式实现了基础设施部分的代码并且预先解决了常见的各种生产问题。每一个问题都被封装在一个或多个框架模块中，每一个框架都为问题所在的领域或问题相关的基础设施依赖提供了一个完整的解决方案。框架模块解决了前面提到的SRE的多个关注重点，如：

- 性能指标的度量和选择
- 请求日志记录
- 流量和负载管理的控制系统

SRE通过构建框架模块来实现这些关注重点的标准解决方案。其结果是，因为该框架已经考虑了正确的基础设施的使用，所以研发团队可以更专注于业务逻辑的开发。

一个框架基本上是预先实现的一套使用一系列软件组件的规范。该框架还可以以一种标准的方式对外提供控制各个组件的功能。例如，一个框架可以提供如下功能：

- 将商业逻辑用一系列完善定义的语义组件进行组织，可以用标准术语来引用。
- 标准监控维度。
- 请求调试日志的标准格式。
- 管理负载抛弃的标准配置格式。
- 描述一个单一的软件服务器的容量，以及如何判断“过载”的方式，这样可以以一致的方式将度量结果反馈给控制系统。

各种框架在一致性和高效性上提供了很多前期收益。它们将开发者

从临时的、定制的黏合和配置各种组件的工作中解脱出来。这些成果常常是相似又不相同的，它们又必须需要SRE人工一个一个地评审。这些框架推动了跨服务的、可重用的生产解决方案，这意味着该框架的用户最终会以相同的通用方式使用服务，配置差异也很小。

Google内部支持数种语言开发的应用，这些框架在所有语言中都可用。虽然框架的不同实现（C＋＋与Java）不能共享代码，但是目标是为同样的功能暴露同样的API、行为、配置和控制组件。这样，研发团队可以选择适合他们需求和经验的语言平台，而SRE仍然可以期待这些应用在生产中的行为相近，可以利用标准的工具进行管理。

新服务和管理优势

这种基于服务框架和通用生产平台、控制界面的结构化方法提供了一系列新的益处。

显著降低运维开销

一个基于框架和强标准而构建的生产平台能够明显降低运维开销，原因如下：

- 它支持代码结构、依赖关系、测试、编码样式指南等的强合规性测试。这个功能还提高了用户数据的隐私度、测试和安全性的合规性。
- 它具有内置的服务部署、监控和自动化服务。
- 它使得管理大量的服务更容易，特别是数目暴增的微服务。
- 部署更快：一个想法可以在数天之内在SRE级别支持的生产环境中全面部署！

设计中自带的通用性支持

Google内部服务数量的不断增长，意味着大部分服务既不值得SRE参与也不会被SRE维护。但是不管怎样，没有完整SRE支持的服务可以使用SRE开发和维护的生产功能。这种做法有效地突破了SRE的编制障碍，让所有团队都能使用SRE所支持的生产标准和工具，以提高Google

所有团队整体的服务质量。而且，所有用框架实现的服务都会自动从对框架模块的逐步改进中受益。

更快、更低的参与成本

这种框架模型可以使得PRR执行更快，因为我们可以依靠：

● 作为框架实现的一部分内置服务功能。

●“接手”服务更快（通常由一个SRE在一个季度内完成）。

● SRE在管理采用框架构建的服务时的认知负担降低了。

这些特性减少了SRE团队在服务交接时的评估和认证工作，同时仍然能保持对服务产品质量的高标准要求。

一种基于共同责任的新型参与模型

最初的SRE参与模型体现了两种选项：完整的SRE支持，或者基本上没有SRE支持。[19]

一个有着公共服务结构、惯例，以及软件基础设施的生产平台，使SRE团队可以为“平台”基础设施提供支持，而让研发团队为服务的功能性问题提供on-call支持—也就是应用程序代码中的错误。在这种模式下，SRE承担大部分基础设施服务的软件开发和维护的责任，特别是控制系统，如负载抛弃、过载保护、自动化、流量管理、日志和监控等。

该模型体现了一个从最初以两种主要方法为主的服务管理方式的重大变革：它需要一个SRE与研发团队之间的新的关系模型，以及一个对于SRE所支持的服务进行管理的新人员配置模型。[20]

小结

服务的可靠性可以通过SRE的参与来改建，该流程包括系统化的评审与生产运维方面的不断改进。Google SRE最初的系统化流程——简单PRR模型，在规范SRE参与模式中取得了长足的进步，但仅适用于已经进入发布阶段的服务。

随着时间的推移，SRE扩展和改进了这一模型。早期参与模型在开发周期早期引入了SRE，以便“为可靠性而设计”。随着对SRE专业知识需求的持续增长，对更具扩展性的参与模式的需求也更强烈了。为了满足这方面的需求，生产服务框架随之出现了：基于生产最佳实践的代码模式在框架中进行了标准化，封装在框架内，使得使用框架成为一个被推荐的、一致的、相对简单的构建服务的方法。

上文描述的这三种参与模型仍然在Google内不断实践。然而，框架正在成为Google内部构建生产完备的服务的主要力量。这也是SRE深入扩展自己的贡献的主要领域，这同时降低了管理成本，提高了整个Google的基础服务水平。

[1]作者是 Laszlo Bock,Google 人力资源部门VP。

[2]这里还包括系统为什么不能正常工作的知识。

[3]主动性SRE的工作包括：软件自动化，架构设计咨询，发布流程协调等。

[4]被动性SRE的工作包括：在线调试，故障排除，处理on-call情况等。

[5]正如游戏里那样！

[6]“跟踪RPC”这种手段同时也适用于批处理/流水线系统：从跟踪启动系统的那个操作开始。对批处理系统来说，这个操作可能是指需要处理的数据到达指定的问题，或者某个需要被校验的交易操作，或者其他的一些事件。

[7]请参考“Life in the Trenches of healthcare.gov”（http://www.thedotpost.com/2014/05/robert-kennedy-life-inthetrenches-of-healthcare-gov）。

[8]例如：“另外一个团队的成员发出了一条警报，他们是这么说的：……”

[9]例如：“我们正在损失钱！如何能够在短时间内缓解这个问题？”

[10]见Wikipedia:Flow（psychology）,http://en.wikipedia.org/wiki/Flow_（psychology）。

[11]注参看http://en.wikipedia.org/wiki/Running_the_gauntlet。

[12]原文中采用的是post-vitam，拉丁文。

[13]并且，如我们所知，文化通常必胜策略，参见文献[Mer11]。

[14]更大型的团队通常抢过小型团队的话头，也经常进行那些分散注意力的谈话等。

[15]在这个案例中，JavaScript的大量使用也是一个很严重的因素。

[16]也就是说，软件和生产软件的组织具有相同的沟通结构，可参考https://en.wikipedia.org/wiki/Conway%27s_law。

[17]DoubleClick for Publisher 是一个给出版商的工具，管理在他们的网站和在他们的应用程序上展示的广告。

[18]参看维基百科上的微服务的页面，http://en.wikipedia.org/wiki/Microservices。

[19]偶尔，SRE团队会为一些未“接手”的服务提供咨询服务，但是咨询服务是没有保障的，在数量和范围上都很有限。

[20]新的服务管理模型以两种方式改变了SRE人员配置模型：（1）因为大量的服务采用通用技术，每个服务所需要的SRE人数也随之减少；（2）它促成了生产平台的建立，使得关注重点划分为由SRE支持的生产平台和由开发团队支持的服务的具体业务逻辑。这些平台团队根据所维护平台的需要而配置人员，而非基于所服务的数量来配置，同时可以在多个产品之间共享资源。

第V部分 结束语

在讨论完Google SRE的工作细节，包括其中的指导思想与最佳实践如何应用到同领域的其他组织之后，我们可以将目光转向第33章，本章将SRE的实践经验与其他同样非常重视可靠性的行业进行了对比。

最后，Google VP,Benjamin Lutch 在全书最后部分回忆了SRE在Google内部的演进历史，同时也将SRE与民航行业的某些实践经验进行了对比。

第33章 其他行业的实践经验

作者：**Jennifer Petoff**

编辑：**Betsy Beyer**

在深入讨论Google SRE的文化和实践时，我们很自然地会联想到其他行业是如何对待可靠性这个问题的。借着编纂本书这个机会，我有幸和许多Google工程师一起讨论了他们之前的工作经历，这些行业也非常注重可靠性。在讨论时，我的目标是回答下列几个问题：

- SRE采用的指导思想是否在Google外部也有采用？其他行业是否在用截然不同的方式解决高可靠性问题？
- 如果其他行业也采用同样的指导思想，那么具体对应的实践是什么样的？
- 不同行业之间的具体实践的相似点与不同点是什么？
- 是什么因素导致了这些相似点和不同点的产生？

- Google以及科技行业可以从这些对比中学到什么？

在本章中，我们会讨论到许多SRE的核心指导思想。为了简化与其他行业最佳实践的比较，我们将这些理念分为4大类：

- 灾难预案与演习
- 书写事后总结的文化
- 自动化与降低日常运维负载
- 结构化的、理智的决策

这一章中同时介绍了有其他行业经验的资深SRE。我们一起讨论了关键的SRE思想，以及这些思想是如何在Google内部应用的，同时也给出了其他行业内部应用这些思想的例子。最后，以一些模式和反模式作为结尾。

有其他行业背景的资深SRE

Peter Dahl（我的直接Boss）是Google Principal Engineer（部门总监或者VP级别）。在之前的职业生涯中，他曾经在美国国防部的一个供应商处任职，负责陆空运载设施的GPS与惯性制导系统。在这种系统中，如果出现一丁点儿不可靠的情况，就会造成整个运载设施故障或者事故，同时也会造成巨大的经济损失。

Mike Doherty是现任 Google SRE。在加拿大，他曾经是一名救生员，也是一名救生员教练，任职十年。可靠性在这个领域也是很重要的，因为每一刻都是人命关天的。

Erik Gross现任Google 软件工程师。在加入Google之前，他花了7年时间设计激光视网膜手术（LASIK）的算法与具体实现。这个领域对可靠性非常关注，压力也是很大的。伴随着这项技术的成熟，FDA批准上市，逐渐改进以及普遍运用，他从中学到了很多关于科技与人在高可靠方面上的经验。

Gus Hartmann与**Kevin Greer**曾经有通信行业的经验，包括维护

E911紧急呼叫系统。[1]Kevin现在是Google Chrome团队的一名软件工程师，而Gus是Google IT团队的一员。通信行业也是一个注重高可靠性的行业，用户体验与可靠性直接相关。如果服务出现问题，不仅仅用户体验受到影响，如果E911系统出现故障，则甚至会有人员伤亡。

Ron Heiby是Google的一名技术项目经理。Ron有汽车、医疗器械，以及手机行业的开发经验。他曾经负责处理这些行业对接组件的开发（例如，某个设备用数字无线网络传输救护车上的EKG数据[2]）。在这些行业里，可靠性方面出现问题是可能导致设备被召回而受到业务损失，甚至会间接导致人身危害的，例如，EKG设备无法与医院通信可能会导致治疗不及时等。

Adrian Hilton是Google的一名发布协调工程师（LCE）。在之前的职业生涯中，他曾经参与设计实现英国与美国的军用飞机、海上飞机、飞机地勤管理系统，以及英国的列车信号系统。这些行业的可靠性极为重要，因为任何一点事故都可能会导致几百万美元的设备损失，以及人员伤亡。

Eddie Kennedy是Google全球用户体验组的一名项目经理，也是一名机械工程师。Eddie 曾经在某个合成钻石工厂供职6年，是Six Sigma黑带、生产流程工程师。这个行业对安全有着不懈的追求，因为生产过程中涉及的温度极高、压力巨大，生产过程对工人有极高的危险性。

John Li目前是Google SRE。John之前在一个财务交易公司任职系统管理员与软件开发者。财务行业对可靠性要求也很高，因为这可能会导致巨大的财务损失。

Dan Sheridan是一名Google SRE。加入Google之前，他曾是英国民用核工业的一名安全顾问。在核工业行业中，可靠性非常重要，因为任何一点问题都可能导致无法挽回的损失：事故可能会导致每天损失数百万美元的收入，同时对员工以及附近居民的威胁更大。这个行业基本对故障零容忍。设计核设施时包含了一系列防故障系统（failsafe），这些防故障系统会在大型事故发生之前终止所有运行，防止事故扩大。

Jeff Stevenson目前是Google的硬件运维团队经理。过去，他曾经

是US海军的一名核潜艇工程师。核动力潜艇的可靠性也非常重要，故障可能会导致设备受损，长期性的自然环境污染，或者人员伤亡。

Matthew Toia是存储系统的一名SRE经理。在加入Google之前，他曾经参与空中交通管理软件系统的开发与部署。该系统如果出现故障，可能会导致旅客出行受到影响，航空公司受损（飞机延迟、改线等），甚至会造成飞机坠毁，人员伤亡。这个行业广泛使用了纵深防御的策略来防止灾难发生。

上述是和我们讨论过的所有行业专家，相信读者也对他们之前的行业对可靠性的关注有了粗略认识，下面我们就详细论述可靠性的4个大类。

灾难预案与演习

“愿望不是一个策略”,Google的SRE的这条口号很好地总结了我们对灾难预案与演习的态度。SRE的文化是永远保持警惕，不停地提出疑问：什么可能出现故障？在故障导致服务停止或者数据丢失之前我们如何避免？我们的年度灾难恢复演习（DiRT）就是为了解决这个问题（参见文献[Kri12]）。在DiRT演习中，SRE在生产环境中创造真正的问题，以便能够：

- 确保系统按我们预想的方式应对故障。
- 寻找系统中未预料到的弱点。
- 寻找其他提高系统鲁棒性的方式来避免事故发生。

在其他行业中，关于如何测试团队的灾难准备情况，以及保障团队应对灾难的能力的策略有如下几个方面：

- 从组织架构层面坚持不懈地对安全进行关注。
- 关注任何细节。
- 冗余容量。

- 模拟以及进行线上灾难演习。
- 培训与考核。
- 超乎寻常地关注对细节要求的收集与系统的设计。
- 纵深防御。

从组织架构层面坚持不懈地对安全进行关注

该理念对工业工程来说格外重要。Eddie Kennedy 曾经在一个高危制造车间工作，随时面临着格外危险的环境，“每次管理会议都是以安全讨论开头”。制造业通过建立极为确定的流程，以及保证整个组织从上至下严格遵守该流程来确保不会出现意外事故。每个员工对安全的关注是很极为重要的，这样每个员工在他们发现出现问题时可以主动提出。在核动力行业、军用飞机，以及轨道交通信号系统行业中，软件的安全标准是明确定义的，例如UK Defense Standard 00-56、IEC 61508、IEC 513、US DO-178B/C、DO-254。同时，这些系统的可靠性级别也都是明确定义的（安全整合级别（SIL）1-4）。[3]这些规定的目标都是为保障交付软件产品的质量。

关注任何细节

Jeff Stevenson回忆起在美国海军的任职经历，所有人对某些小任务执行过程中出现的粗心情况可能会导致大型潜艇事故的情况非常了解（例如，润滑油的及时补充）。非常小的一个错误都可能产生极为严重的后果。系统相互连接紧密，所以一个区域的事故可能会导致多个相关系统出现故障。核动力海军对日常常规维护非常重视，以确保小问题不会发展成大事故。

冗余容量

系统利用率在通信行业中可能是非常难以预知的。系统容量可能会被无法提前预料的事件所影响，例如自然灾害等，又或者是大型的可预知的活动，例如奥林匹克运动会。在Gus Hartmann的从业经历中，通信行业利用SOW（switch on wheels）——一个移动通信平台来提供冗余

容量，以便应对这些情况。这些额外的容量可以应急部署，也可以在某个大型事件发生之前提前部署。同时，系统容量不仅是指系统的绝对容量，也和其他方面有关。例如，某个明星的电话号码在2005年被泄露之后，数以千计的粉丝同时试图拨打这个电话，整个通信系统陷入了一种类似DDoS的状态，造成了大面积的路由错误。

模拟以及进行线上灾难演习

Google的灾难恢复团队在模拟与线上灾难演习方面与其他行业的关注点非常类似。利用某种灾难情景可能导致的故障的严重程度来决定是使用模拟方式，还是线上方式进行演习。例如，Matthew Toia指出，航空行业由于设备和人员安全因素无法进行真正的线上测试。他们则部署了极为逼真的模拟器，这些模拟器使用线上真实数据。在测试中，控制室与设备在细节上和真实环境高度一致，以确保可以得到真实的测试数据。Gus Hartmann同时提到，通信行业一般采用线上演习模式，主要关注于应对飓风或者其他环境灾难。这种理念使得很多通信行业都建设了防水机房，同时内部带有可以运行超过飓风持续时间的发电机。

美国核动力海军会同时进行假想练习与真实的实战演习。Jeff Stevenson说，实战演习包括“在可控环境下进行破坏”。实战演习每周进行，每次持续两到三天。对核动力海军来说，假想练习是有用的，但是却不能真正防止灾难发生。灾难的响应机制必须要经过不断练习才能确保不会忘记。

Mike Doherty，曾经的救生员，这个行业采用一种类似“神秘顾客”的方式进行灾难演习。一般来说，设施负责人会和一个儿童，或者是某个正在接受训练的救生员一起构造一起伪落水事故。这些情景会制造得非常逼真，救生员一般无法区分真实和虚构的紧急事故。

培训与考核

我们的座谈结果显示，培训与考核在涉及人身安全的行业极为重要。例如，Mike Doherty描述了救生员如何通过非常严谨的培训考核过程，同时还要定期接受再考核。培训课程包括健身部分（救生员必须能够将某个比自己还重的人托住，使得他们肩膀露出水面），也包括技巧

部分（例如急救和CPR），同时还包括日常流程方面（例如，当某个救生员跳入水里时，其他团队成员应该做什么）。每个设施都有当地特有的培训课程，因为在游泳池里救人和在湖边或者海中救人有很大差异。

对详细的需求收集和系统设计的关注

与我们座谈的一些工程师讨论了详细的需求收集与设计文档的重要性。这些实践对医疗器械来说尤其重要。在很多案例中，某个器械的使用情况或者维护情况与设计者的预想很不一样。因此，具体的使用情况与维护需求必须从其他来源获取到。

例如，根据Erik Gross的经验，激光视网膜手术设备被设计得极为简单明了，确保使用它们的人不会出错。因此，从真正使用这些设备进行手术的医生处，以及维护这些设备的工程师处获取需求是很重要的。在另一个例子里，Peter Dahl描述了国防系统中对设计高度重视的文化。构建新的国防设备甚至需要一整年的设计过程，最后的实际编码过程只需要三周。这两个例子与Google的发布迭代文化截然不同，我们更倾向于在计算过的风险性前提下，最大化迭代的速度。其他行业（如上文提到的医疗行业和军用行业）对风险的承受能力与我们区别很大，这就导致了他们采用的方案与Google截然不同。

纵深防御

在核能行业中，纵深防御是灾难预案中的一个关键元素（参见文献[IAEA12]）。核反应堆的所有系统都有冗余备份，同时在设计中，每个主要系统都强制备有后备系统。整个系统设计时加入了多层防护机制，包括最后的发电站核辐射物理屏障。纵深防御理念在核能行业中非常重要，因为他们对事故和灾难几乎是零容忍的。

事后总结的文化

纠正性和预防性操作（CAPA）[4]是提升可靠性的一个常见概念。这意味着系统化关注于引起故障，或者带来风险的根源问题。这与SRE“对事不对人”的事后总结理念非常一致。当出现问题时（由于Google内部环境的海量规模、复杂程度，以及非常高的变化速度，一定会有东西

出现问题），详细考虑下列问题是很重要的：

- 究竟发生了什么
- 响应的有效程度
- 下次是否可以采用其他方案解决问题
- 如何确保这次故障不会再次发生

这个过程在执行的时候不应该归责于任何一个个人，更重要的是，作为一个组织我们要找出什么出现了问题，以及如何确保问题不再发生。纠结于“谁”造成了这个故障是没有意义的。事后总结在每次事故发生之后都会进行，同时会在整个SRE团队内部传阅，以便让所有人都能从中受益。

我们的座谈发现了很多行业也进行类似的事后总结活动（但是很多都不会用Postmortem这个词语来指代，因为这个词有验尸的意思）。但是，在不同的行业内部，他们进行事后总结的原动力各有不同。

很多行业都受到政府部门的严密监管，出现问题时，会有相关的政府部门追责。这样监管在灾难预期后果严重的时候是更加紧密的（例如那些会造成人员伤亡的事故）。相关的政府监管部门包括 FCC（通信行业）、FAA（航空业）、OSHA（制造业与化工业）、FDA（医疗器械），以及 欧盟中的数个NCA。[5]核工业与运输业也被严密监管。

另外一个进行事后总结的动力是对安全的关注。在制造业与化工行业中，由于生产过程的危险性，员工伤亡的风险是一直存在的（高温、高压、有毒气体、腐蚀性液体等）。例如，Aloca在安全问题上非常关注。前CEO Paul O'Neil要求员工在出现人员伤亡事故的24小时内通知他。他甚至将自己家的电话号码直接分发给员工，这样一线员工可以在发现安全问题时亲自联系他。[6]

制造业与化工行业的危险性非常高，以至于“差点出事”—某个事件可能会造成严重伤亡，但是没有—这样的事件也会被详细追究。这些场景成为了一种预防性的事后分析。根据 VM Brasseur在YAPC NA 2015

的一次演讲，“在任何事故和业务灾难发生之前，类似事故都曾经发生过好几次，只是没有造成任何后果。这些事故在发生的时候都被忽略了。潜伏的错误，加上某个适合的时机，就会导致事故的发生。”（参见文献[Bra15]。）这种没出事的事故事实上就是一种潜伏的事故。例如，某个工人没有遵守标准的运营流程，或者某个工人在出现泄漏的时候躲开了，或者是台阶上的液体没有及时清理。这些都意味着潜在事故的发生，以及学习和优化的机会。下一次，整个公司和员工可能就不会这么幸运了。英国的CHIRP（航空与航海保密汇报机制）允许航空和航海行业的从业人员匿名汇报这类事故的发生，以在整个行业内部敲响警钟。关于这类差点出事的事故报告和分析会在周期性的杂志上刊登。

救生员行业也有一个根深蒂固的事后分析与计划文化。Mike Doherty 讽刺道：“如果救生员的脚踏入了水里，后续就要有很多报告要写。”在游泳池或者沙滩上出现的任何事故都需要一个详细的事后总结。对于严重的问题，团队还会集体从头到尾分析案例，讨论哪些行动是正确的哪些是错误的。接下来会根据这些讨论结果修改运营规则，同时还会组织培训帮助大家建立处理类似事故的信心和能力。在某次特别严重的或者创伤性的事故发生之后，甚至会有医疗顾问来帮助团队从精神创伤中恢复。救生员可能已经尽力了，但是还是会感觉到他们做得不够。和Google类似，救生员行业也采用了对事不对人的事后分析机制。事故发生时总是非常混乱的，许多因素都可能影响事故的发生。在这个领域中，指责某一个具体的人是没有意义的。

将重复性工作自动化，消除运维负载

Google SRE本质上还是软件工程师，他们对重复性的、被动性的工作十分反感。在我们的文化中强调避免反复执行一项重复性的工作。如果一项工作可以自动进行，为什么还需要我们来重复执行呢？ 自动化可以降低运维负载，同时节省工程师的时间。他们可用这些时间去主动改进服务的其他方面。

我们所调研的各个行业在对待自动化的问题上各有不同。某些行业信任人多过自动化。美国核动力海军通过一系列交叉管理流程来避免自动化。例如，根据Jeff Stevenson的经历，操作某个阀门需要一个操作

员、一个监督员，以及一个船员与负责监控该操作的工程负责人保持通话。这些操作过程是极为人工化的，因为担心自动化系统可能不会发现人能注意到的问题。潜艇上的操作是由一条信任的人工决策链管理的——一系列人，而非单独一个人。核动力海军同时担心自动化和计算机运行速度太快，以至于它们可能造成非常严重且无法挽回的问题。当面对核反应堆时，缓慢而稳健的方法比快速完成任务更重要。

根据 John Li的经验，私有化交易行业在近年来对自动化的应用越来越小心。经验证明，配置错误的自动化系统可能会在极短的时间内造成极大的财务损失。在2012年，Knight Capital Group 遇到了一个“软件错误”，在几个小时之内造成了4.4亿美元的损失。[7]同样的，美国股票交易所2010年经历了一次闪电崩溃（Flash Crash），最终归责于一个交易商试图利用自动化方式操纵股市。虽然股票市场很快恢复了，但是这次闪电崩溃在30分钟内造成了千亿美元的损失。[8]计算机能够非常快地执行操作，在这些操作配置错误的时候，速度反而是有害的。

相反，某些公司正是因为计算机比人要反应快得多而大量采用自动化。根据Eddie Kenned的经验，效率与成本的节约在制造业中是非常关键的。自动化可以以更高的效率、更低的成本来完成这些工作。而且，自动化通常比人工完成工作的重复性更强、更可靠，这意味着自动化可以产出更多高标准的产品。Dan Sheridan讨论了部署于英国核能工业中的自动化机制。这里，采用的标准是如果整个发电站需要在少于30分钟的时间内响应某种情况，那么这种响应必须要自动化进行。

在Matt Toia的经验中，航空行业有选择地采用自动化。例如后备系统的切换一般来说是自动进行的，但是在其他任务上，行业内部普遍要求人工进行二次校验。虽然整个行业内部采用了很多自动监控系统，实际的空中交通管理系统最终的实现还是需要人来校验运行。

在Erik Gross的故事中，自动化机制在降低激光视网膜手术过程中的用户错误是很有效的。在LASIK手术进行之前，医生需要进行屈光度测试。在最初的设计中，医生需要手动输入数字，再按一个按钮，激光系统就会开始视力矫正过程。然而，数据输入错误的问题很严重。在这个流程中，可能会错误输入另外一个病人的数据，甚至是左右两只眼睛

的数据搞混了。

自动化在这个环节中减少了这种错误带来影响的可能性。一个自动化的对输入数据的正确性检查是第一个改进：如果操作者输入了超出预期范围的数字，自动化程序会立刻自动提出警告。其他自动化的改进包括：操作环节中的虹膜照片会自动与屈光度测量环节的虹膜图像进行对比，以防止数据混乱情况发生。当这种自动化解决方案最终实现时，这一类的问题就全部消失了。

结构化和理性的决策

在Google内部，尤其是SRE部门内部，数据是极为重要的。整个团队通过以下几种方式保障结构化和理性的决策过程：

- 某项决策的基本方向是事先决定的，而不是事后得出的。
- 决策时考虑的信息源是清楚的。
- 任何假设都应该明确说明。
- 数据驱动决策要优于情感驱动的决策、直觉驱动的决策，以及资深人士的意见。

Google SRE同时用以下信息作为团队内部的基础要求：

- 每个人都为服务的用户负责。
- 每个成员都能够根据可用的数据找出执行方案。

决策应该是通知式的，而不是指令式的，决策的过程不应该考虑个人意见—哪怕是最资深的成员。Eric Schmidt和Jonathan Rosenberg将之称为“HiPPO”—工资最高的员工的意见（参见文献[Sch14）]。

其他行业中进行决策的过程有很大不同。我们了解到，某些行业使用“如果没有坏，就永远不要试着修复它”这种理念来进行决策。这些在设计过程中投入很多思考与精力的行业有一个显著的特点—非常不愿意改变底层技术。例如，通信行业仍然在使用19世纪80年代部署的远程交

换机。为什么他们要依赖几十年前的技术呢？因为这些交换机“基本上不会出错，同时冗余度非常高”，Gus Hartmann解释道。Dan Sheridan 提到，核能行业也类似。所有的决策都由一个想法驱动：“如果目前运行正常，那就不要修改它。”

很多行业都非常依赖手册与流程，而不是实时的故障排除。任何人类能想到的故障场景都在一个检查列表中或者“大本子”里面记载了。当故障发生时，这些资料是进行操作的权威指南。这种指令式的方式适用于那些发展和演进较慢的行业，因为故障场景不会因为系统升级而快速变化。这种方式同时适用于那些员工技能水平较低的行业，这样可确保员工在紧急情况中正确响应的方法就是提供一套简单清晰的指令集。

其他行业也会采用很清晰的、数据驱动的决策方式。在Eddie Kennedy的经验中，制造业中的一大特点就是试验文化，对构建和测试猜想非常关注。这些行业经常性地进行某种可控的测试，保证某种变更不会导致结果的大幅度改变，确保没有意外情况发生。变更只有在经过实验证明后才会实施。

最终，某些行业，例如交易行业，将决策划分成更小的块来更好地管理风险。根据John Li的经验，交易行业有一个专门的风险控制部门，独立于交易部门之外，负责确保不会在追求利益的同时承受不必要的风险。这个风险控制部门负责监控交易场所内部，同时在交易发生意外的时候终止。如果某个系统发生异常情况，风险控制部门的第一反应是关掉这个系统。John Li提到：“如果我们不进行交易，那么我们就不会亏钱。虽然我们也没有赚钱，但是起码不会亏钱。”只有风险控制部门可以将系统重新启用，不论交易员是否错失了某种可以赚钱的机会。

小结

Google SRE的很多核心思想在很多行业中都得到了验证。这些成功行业的很多经验教训可能启发了一些Google今天内部所使用的流程与实践。

在本次跨行业调研中最主要的一点结论是，与很多软件行业相比，Google明显更倾向于变更的速度。但是在追求变更速度的同时一定要考

虑到故障发生时的影响。例如在核动力、民航，以及医疗行业中，如果造成事故可能会有人身伤亡。当故障的影响这么大时，采用更保守的方法是必要的。

在Google内部，我们经常在用户对高可靠性的预期与内部对快速变更和创新的追求之间寻求平衡。虽然Google对可靠性非常看重，我们也必须要适应内部快速的变更频率。正如前面几章提到的，我们的很多业务（例如搜索）都要在产品层面进行决策，来决定到底多么可靠才是“足够可靠”。

Google的软件产品和服务如果出现问题，并不会直接造成人身伤亡，这是一个优势。因此我们可以利用例如错误预算这样的工具来支撑快速创新的文化。总的来说，Google已经采用了行业内部普遍采用的可靠性原则，并且创建了自己独特的文化。Google的这种文化可以满足内部对扩展性、复杂度、迭代速度以及高可靠性之间追求的平衡。

第34章 结语

作者：**Benjamin Lutch**[9]

编辑：**Betsy Beyer**

我是带着极大的自豪感读完本书的。我在1990年左右加入Excite，负责 “软件运维团队”—算是SRE的祖先。我的整个职业生涯就是围绕着构建一个高可靠、高可用的系统的。看到SRE这个想法在Google生根发芽是一件非常令人激动的事情。SRE从2006年我刚加入Google时的近100个工程师发展到今天的近1000人，横跨十几个分公司。可以说，SRE目前负责运维的是整个地球上最前沿的计算基础设施。

那么，究竟Google SRE是靠什么来维持10年以来大规模部署的基础设施的效率与扩展性的呢？我认为SRE这种惊人的成功主要是靠其日常工作中的一些核心指导思想。

SRE团队在结构设计上要求工程师将时间平均分配在两种同等重要

的工作上。SRE负责参与on-call轮值，这需要SRE动手实际运维系统，观察和调整系统的弱点，以及理解如何能大规模扩展这些系统。但是我们同时也需要时间反思与决策如何才能够让这些系统更容易管理。从某种程度上来说，我们既是飞机驾驶员，也是飞机的设计者和工程师。我们运行海量系统的经验都固化成了实际的代码，同时也包装成了对外提供的产品。

这些解决方案可以很容易地被其他SRE团队重用，甚至提供给Google之外的人（利用Google Cloud等服务）。任何人都可以使用甚至优化这些年SRE积累的经验与构建的系统。

当组建团队或者构建系统时，最理想的情况是制定一系列通用的基本规则和公理。这些规则与公理应该是足够通用并立即可以采用的，也应该能够在未来保持一定的相关性。Ben Treynor Sloss在本书第一章正是这么介绍SRE的：他制定了一系列灵活的，维持十年不变的SRE根本职责。虽然十年中Google的基础设施有了长足发展，SRE团队也大不一样了，但是这些职责仍然没变。

在SRE的发展中，有很多截然不同的、相互作用的因素在相互推进。首先，SRE的职责与主要关注重点在十年内基本保持不变：虽然系统部署规模可能扩大了1000倍，系统性能可能提升了1000倍，但是这些系统仍然需要保持灵活，能够应对紧急情况，监控周全，做好容量规划等。其次，SRE的日常工作随着服务的发展以及SRE团队的成熟在不断变化。例如，最初的“构建一个监控20台服务器的监控台”可能逐渐演化成“为几万台服务器构建能够自动服务发现、自动生成监控台与报警”这样的任务。

对那些不是本行业的人来说，可以用民航行业在过去十年中对飞机飞行过程的管理的变化来比喻SRE对复杂计算系统的管理。虽然这两个行业中事故所造成的后果相差很远，但是基本的核心理念是一致的。

假设100年前的我们需要在两个城市之间飞行。我们的飞机可能是单引擎的飞机（如果特别幸运的话，是双引擎的），飞机上装载了一些货物，以及一个飞行员。飞行员同时也担当了机械师的角色，甚至可能还要负责货物的装载和卸载。飞行驾驶舱中一般只有飞行员的位置，最

好的情况下还有一个副飞行员或者是导航员的位置。每隔几天，我们的飞机就会在晴朗的天气中起飞，慢慢爬升高度，最终降落于另外一个几百公里之外的城市，以此往复。在这个过程中，任何故障的后果都将是灾难性的，更不要说飞行员可以在飞机过程中爬出驾驶舱修理飞机了！飞机上接入驾驶舱的所有系统都是非常必需的，简单的，但也是易坏的，基本都没有任何冗余度的存在。

经过100年来的变迁，我们来看一下现在停在停机坪上的某架巨型747飞机。数百名旅客同时从多层进入飞机，数吨重的货物同时进入底下的货舱。飞机上布满了非常可靠的、冗余度非常高的系统。这就是不断重视安全与可靠性的后果：事实上，目前比起在路上开车，在空中飞行要更安全！现在的飞机会在某个大洲起飞，轻松地在另外一个6000英里以外的大洲降落。严格遵守飞行时间，基本在预计时间前后几分钟内到达。但是我们再来看一下驾驶舱，仍然只有两名飞行员！

为什么飞行过程中的所有其他元素—安全性、容量、速度、可靠性—都发生了翻天覆地的变化，而仍然只有两名飞行员呢？这里的答案与Google SRE运维大规模、超复杂的系统的方法是一致的。飞机操作系统的人机接口都是精心设计过的，简单易用、易于学习，使得飞行员可以正常操纵飞机飞行。同时，这些界面又提供了足够的灵活性，飞行员也经过了大量训练，使得他们可以很快、很可靠地应对紧急情况。飞机驾驶舱是由理解复杂系统的人设计的，他们知道如何用一种可扩展的、易于理解的方式给飞行员提供数据。飞机中的系统都具有很多本书中提到的各种特性：高可用性、性能极度优化、变更管理、监控与报警、容量规划，以及应急处理。

最终，SRE的目标也是如此。SRE团队应该越精简越好，他们所操作的东西应该更抽象而非更具体。SRE团队依赖各种后备系统和精心设计的API来运维。同时，SRE也应该对系统的运作原理、运维方式、故障模式以及应急处理方式非常了解—这些都是在日常工作中学到的。

[1]E911（Enhanced 911）：美国内部使用的地理位置信息的紧急报警系统（类似中国的110）。

[2]更多信息可参见https://en.wikipedia.org/wiki/Electrocardiography。

[3]参见https://en.wikipedia.org/wiki/Safety_integrity_level。

[4]参见https://en.wikipedia.org/wiki/Corrective_and_preventive_action。

[5]参见https://en.wikipedia.org/wiki/Competent_authority。

[6]参见http://ehstoday.com/safety/nsc-2013-oneill-exempli☆es-safety-leadership。

[7]有关Knight和Power Peg Software的讨论见“FACTS,Section B”（文献[Sec13]）。

[8]参见“Regulators blame computer algorithm for stock market‘flash crash',”Computerworld,http://www.computerworld.com/article/2516076/☆nancial-it/regulators-blame-computer-algorithm-for-stock-market—flashcrash-.html。

[9]Google VP,SRE团队负责人。

附录A 系统可用性

系统可用性通常是以某段时间内服务不可用的时间比例来计算的。假设没有任何计划内停机时间，表A-1描述了每个可用性级别所允许的停机时间。

表 A-1：可用性时间表

水平	允许的不可用窗口时间					
	每年	每季度	每月	每周	每天	每小时
90%	36.5 days	9 days	3 days	16.8 hours	2.4 hours	6 minutes
95%	18.25 days	4.5 days	1.5 days	8.4 hours	1.2 hours	3 minutes
99%	3.65 days	21.6 hours	7.2 hours	1.68 hours	14.4 minutes	36 seconds
99.5%	1.83 days	10.8 hours	3.6 hours	50.4 minutes	7.20 minutes	18 seconds
99.9%	8.76 hours	2.16 hours	43.2 minutes	10.1 minutes	1.44 minutes	3.6 seconds
99.95%	4.38 hours	1.08 hours	21.6 minutes	5.04 minutes	43.2 seconds	1.8 seconds
99.99%	52.6 minutes	12.96 minutes	4.32 minutes	60.5 seconds	8.64 seconds	0.36 seconds
99.999%	5.26 minutes	1.30 minutes	25.9 seconds	6.05 seconds	0.87 seconds	0.04 seconds

对一个可能部分保持可用的服务，或者是每天、每周负载变化很大的系统来说，利用一个汇总过的不可用指标（例如，“所有操作中的X%失败了。”）来思考可用性更有意义一点—这些系统可能有多个副本，只有某一个副本不可用。

更详细的计算过程请参见第3章中的公式3-1与公式3-2。

附录B 生产环境运维过程中的最佳实践

作者：Ben Treynor Sloss

编辑：Betsy Beyer

可控的故障模式

配置文件，或者服务的任何输入都应该经过正确性和准确性检查再提供给服务。服务在接收到不合理的配置文件或者输入数据时，应该继续保持之前的状态正常工作，同时发出错误输入的警报。错误的输入数据一般分为以下几类：

不正确的数据

在处理数据之前，应该检查数据的语法，甚至在可能的情况下，检查数据语义的正确性。同时，服务应该注意空数据、部分数据或者截断数据的可能性（服务应该在新数据比之前的数据小N%的时候发出警报）。

过期的数据

这些数据可能会影响到目前的数据。应该在数据过期之前发出警报。

服务应该追求在出现故障时仍能够保持工作，可能会牺牲一定程度的访问控制能力，或者使用简化逻辑。我们发现，最安全的方式是在服务收到新数据之后，仍然维持之前的配置运行，直到某个人来批准采用新数据—这些数据可能是无效的。

示例

2005年，Google全球DNS负载/延迟均衡系统由于文件权限错误的原因，收到了一份空数据。该服务接受了空配置文件，为全部Google DNS请求返回了NXDOMAIN结果，故障持续了6分钟。在之后的解决方案中，系统会针对新的配置文件进行一系列安全性检查，包括对google.com IP地址的确认。该系统在收到新配置文件未能通过校验的情况下会一直按之前的配置文件工作。

2009年，一份错误的（但是格式正确的）数据导致Google将整个Web都标记成病毒感染网站（参见文献[May09]）。一份含有病毒URL地址的配置文件内容被错误地替换为只有“/”，导致与全部URL都匹配。后续的解决方案中包括：检查配置文件的大幅度改变，检查配置文件是否与某些已知不太可能含有病毒的网站匹配，这样可以避免这种事故的发生。

渐进式发布

非紧急的发布过程应该是按阶段进行的。不管是配置文件改变，还是二进制文件改变，都会引入一定的风险，我们通过小规模地应用这些变更来控制这些风险。每次发布部署的容量百分比，以及每个阶段之间等待的时间应该由服务的规模、发布的规模，以及服务所能承受的风险来决定。同时，每个部署阶段中包括多个地理位置也是好主意，这样可以更快地检测到由于流量峰谷或者不同地理区域的流量带来的问题。

整个发布过程应该是有监管的。为了确保发布过程中没有未预料的情况发生，工程师—或者是一个可靠的监控系统—应该对发布过程进行监控。如果出现了意外情况，回退是第一选择，后续再进行详细的分析，这样可以降低平均恢复时间（MTTR）。

从用户的视角来定义**SLO**

应以最终用户的视角来定义可用性与服务性能的SLO，详细的讨论参见第4章。

示例

在Gmail客户端中度量错误率与延迟，而不是从服务器端度量。这样做使我们对Gmail可用性的评估有显著的影响，这直接导致了Gmail客户端与服务器端代码的很多改动。这样做的结果是，Gmail在几年的时间内将可用性从99.0%提升到了99.9%。

错误预算

使用错误预算这个概念来平衡可靠性与创新速度的关系（参见第3章，“使用错误预算的目的”一节）。错误预算定义了某个服务在一段时间内的稳定性目标，我们通常以一个月为单位。预算是通过1-SLO得出的。例如，某个99.99%可用性目标的服务拥有0.01%的错误预算。只要该服务没有在这个月内由于背景错误率和任何停机问题用光这0.01%的错误预算，开发团队就可以（合理地）发布新功能、新变更等。

如果错误预算用完了，服务应该停止任何改动（除了那些非常重要的安全性问题，或者针对错误的Bug修复），直到下一个月。对SLO超过99.99%的成熟的系统来说，按季度计算而不是按月计算可能更合适，因为每个月的不可用时间可能太少了。

错误预算为SRE与产品研发部门建立起一套共同的数据驱动的发布风险评估体系，这样消除了两个部门之间的组织架构矛盾。同时，它也鼓励两个团队共同投入研发新的技术与实践，以便在快速创新和发布的同时避免“用完预算”。

监控系统

监控系统应该仅仅有下列三种输出：

紧急警报

某个人必须执行某项操作。

工单

某个人必须在几天之内执行某种操作。

日志

没有人会马上看这些日志，但是以后需要的时候可以用来分析。

如果某个事件足够重要，需要人的参与，那么它应该需要某种立即动作（也就是，紧急警报）或者是被当作Bug或工单来跟踪。将警报用E-mail发送，希望某些人能够全部阅读它们并且注意到哪些问题是重要的。最终这些警报一定会被无视。历史证明，虽然这种策略在一开始可行，但是最终需要人随时保持警惕性，这会导致最终更严重的灾难的发生。

事后总结

事后总结（参见第15章）应该是对事不对人的，关注流程、技术而非人。假设参与处理事故的人都是理智的，善意的，并利用当时所有可用的信息做出最佳的决策。与这个理念相对应的是，我们不能“修复”人，但是我们可以修复这些人的工作环境：例如，优化系统设计从而避免某一类问题的发生，让合适的信息更可用，自动校验运维操作使得系统进入危险状态更困难。

容量规划

服务应该准备承受计划内和计划外的故障发生，同时不会严重影响到用户体验。这就意味着服务需要“N＋2”配置：在两个最大的副本不可用的情况下，流量峰值可以由N个实例承担（可能是在某种降级模式下）。

- 持续地将事先所做的流量预测与真实情况进行对比，改进预测手段直到它们一致为止。预测与现实出现的偏差可能意味着预测方法不稳定，容量使用效率问题，甚至会导致容量不足。

- 利用负载测试来建立资源与容量的比例，而非依赖历史数据：某个有X台物理机的集群在三个月之前可能可以承受每秒Y个请求，但是随着系统的改变，还能做到吗？

- 不要将上线第一天的负载与稳定状态下的负载混淆。发布通常会吸引到更多的流量，但这恰恰也是需要显示服务质量的时机。更多知识参见附录E和第27章。

过载与故障

服务在过载情况下应该仍然提供合理程度上的次优结果。例如，Google 搜索服务会在过载情况下只搜索一部分索引数据，同时中止某些即时提示功能等，从而能够提供足够优质的结果数据。搜索SRE会对集群进行超出集群容量的测试，确保它们可以在过载情况下正常工作。

当负载高到一定程度时，次优结果也不够的情况下，服务应该采用队列、动态超时等手段进行优雅地流量抛弃，参见第21章。其他的技术包括：将回复延迟发送，或者指定某些用户接受错误，从而保障其他客户的服务质量等。

重试可以将低错误率转化为高流量，从而导致连锁故障（参见第22章）。一旦负载超过容量，就通过丢弃一部分系统上游的流量（包括重试）来应对连锁故障。

每个发送RPC的客户端都应该实现指数型延迟重试（包括抖动），这样可以降低错误放大率。移动客户端问题尤其严重，因为可能有几百万个客户端，修复他们代码中的问题需要花费很多时间，甚至数周，还需要用户安装更新。

SRE团队

SRE团队应该在运维工作上花费不超过50%的时间（参见第5章）；运维过载应该由产品研发部门承担。很多服务会在没有过载的情况下，也让研发团队参与on-call轮值与工单处理。这种做法可以激励研发团队在设计时最小化系统的运维需求，同时也保证了产品开发者熟悉系统的运维过程。SRE与开发团队定期举行常规的生产环境会议也是很有用的（参见第31章）。

我们发现，一个on-call团队至少需要8个人，这样才能避免警报疲劳，确保整个团队的可持续性发展。更好的是，on-call团队最好分布在两个相隔很远的地理位置区域（例如，美国加州以及爱尔兰），这样可以避免某个团队晚上处理紧急警报。在这种情况下，每个区域需要至少6个人。

每次on-call轮值应该处理不超过两起事故（平均每12小时1个）：应对和修复事故都需要时间，书写事后总结以及处理Bug也需要时间。事故频繁会造成响应质量的下降，同时也意味着系统设计中存在一个或多个设计缺陷，监控系统过于敏感，之前书写在事后总结中的问题没有及时修复等。

具有讽刺意味的是，如果我们应用了这些最佳实践，SRE团队最终可能会由于系统事故出现的越来越少而失去熟练性，这可能会造成原本很小的事故需要花很长的时间来解决。通过进行周期性的假想灾难演习可熟悉应对紧急事故的文档与流程。

附录C 事故状态文档示范

莎士比亚搜索服务 新韵文＋＋过载事故：2015-10-21

事故管理网站：http://incident-managment-cheat-sheet

（沟通负责人会随时更新事故概要）

摘要：莎士比亚搜索服务由于新发现的韵文不在索引中而处于连锁故障状态

状态：活跃，事故编号 #465

事故处理中心：IRC #shakespeare 频道

事故处理组织架构：（参与人）

- 目前事故总负责人：jennifer

– 运维负责人：docbrown

– 计划负责人：jennfier

– 沟通负责人：jennifer

- 下一个事故总负责人：待定

（沟通负责人在交接班时或者每4个小时更新一次）

细节状态：（最终更新时间 2015-10-21 15:28 UTC,Jennifer）

退出条件：

- 向莎士比亚搜索服务的Search Corpus中添加新的韵文（TODO）

• 在30分钟内维持SLO，可用性为99.99%，延迟为99%□100ms（TODO）

代办列表以及已提交的工单：

• 执行MapReduce任务，重新索引Shakespeare corpus（DONE）

• 借用一些紧急资源来提高容量（DONE）

• 启用flux capacitor，在集群之间均衡负载（Bug 5554823）（TODO）

事故时间线（倒序排列，时区为UTC）：

● 2015-10-21 15:28 UTC jennifer

– 全球服务容量提升为2倍。

● 2015-10-21 15:21 UTC jennifer

– 将所有流量导向USA-2泄洪集群，同时将其他集群下线，以便让这些集群从连锁故障中恢复，同时启动更多任务。

 MapReduce 索引任务完成，等待Bigtable复制到所有集群。

● 2015-10-21 15:10 UTC martym

– 向Shakespeare corpus中增加新的韵文，同时启动MapReduce索引任务。

● 2015-10-21 15:04 UTC martym

– 从 Shakespeare-discuss@ 邮件列表中获得了新发现的韵文全文。

● 2015-10-21 15:01 UTC docbrown

– 由于出现了连锁故障，声明目前进入紧急状态。

● 2015-10-21 14:55 UTC docbrown

- 出现大量紧急报警，全部集群出现ManyHttp500s。

附录D 事后总结示范

莎士比亚新韵文事故总结（事故编号#465）

日期：2015-10-21

作者：jennifer、martym、agoogler

目前状态：已经终稿，待办事项正在进行中

摘要：莎士比亚搜索服务出现66分钟的故障，由于新发现了一篇韵文导致用户流量暴涨。

事故影响[1]：预计12.1亿个请求丢失，没有损失任何收入。

根源问题[2]：由于异常的高负载情况以及搜索词语在Shakespeare Corpus中不存在时的一项资源泄露导致了连锁故障的发生。新发现的韵文使用了一个之前从未在莎士比亚文献中出现的词语，这恰恰是用户大量搜索的关键词。在日常情况下，这种资源泄露导致的任务崩溃现象由于出现非常不频繁而没有被注意到。

触发条件：潜伏性的Bug被大量上涨流量所触发。

解决方案：将流量导向泄洪集群，同时增加了10倍容量以应对连锁故障。部署了更新过的索引，绕过了潜在的Bug。在公众对新韵文的兴趣消退之前，保持额外的容量。资源泄露的问题已经被找到，并且修复已经上线。

检测：Borgmon检测到大量HTTP　500的情况，向on-call发送了紧急警报。

待办事项：[3]

待办事项	类型	负责人	Bug
更新运维手册（Playbook），加入有关应对连锁故障的指示	缓解	jennifer	n/a DONE
利用 flux capacitor 在多个集群中均衡负载	预防	martym	Bug 5554823 TODO
在下次 DiRT 练习中进行连锁故障演习	流程	docbrown	n/a TODO
调查是否可以持续性地进行索引更新	预防	jennifer	Bug 5554824 TODO
在搜索排序子系统中修复文件描述符的泄露	预防	agoogler	Bug 5554825 DONE
给莎士比亚搜索服务增加流量抛弃功能	预防	agoogler	Bug 5554826 TODO
增加回归测试，保障系统在收到致死请求时保持正常	预防	clarac	Bug 5554827 TODO
将更新过的搜索排序子系统部署到生产环境	预防	jennifer	n/a DONE
由于错误预算耗尽，停止更新生产环境直到2015-11-2。或者寻求管理层对特殊情况的批准	其他	docbrown	n/a TODO

经验教训

做得好的地方：

- 监控系统在大量HTTP 500（接近100%）的情况下快速发出了紧急警报
- 在所有集群中快速更新了 Shakespeare Corpus

做得不好的地方：

- 对连锁故障的处理不够熟练
- 由于这次大幅度的流量增长（几乎全部请求都失败了），导致我们超过了可用性错误预算数个数量级

幸运的因素[4]

- 莎士比亚迷的邮件列表里刚好有一份新的韵文文本
- 服务器日志中包括了指出文件描述符耗尽问题导致崩溃的栈跟踪
- 致死请求（query-of-death）通过推送新的索引关键词就解决了

时间线[5]

2015-10-21（所有时区都是UTC）

- 14:51　新闻报道新的莎士比亚韵文在一辆Delorean车的杂物箱中发现。

- 14:53　/r/shakespeare（指Reddit.com，美国新闻网站）的一篇文章指出莎士比亚搜索引擎是找到新韵文的好地方，流量提升了88倍（这时候我们还没有收录该文章）

- **14:54** 事故开始，搜索后端在高负载情况下崩溃

- 14:55　docbrown　收到了大量紧急警报，所有集群都在汇报ManyHttp500s

- 14:57　所有的莎士比亚搜索请求都在失败：见http://monitor/shakespeare?endtime=20151021T145700

- 14:58 docbrown开始调查问题，发现后端任务崩溃速度很快

- **15:01** 应急故障管理开始，docbrown 宣布事故#465开始，目前处于连锁故障状态，开始在#shakespeare频道进行协调，将jennifer任命为事故总负责人

- 15:02　某人意外发送了一封邮件给Shakespeare-discuss@邮件列表，讨论新的韵文，这封邮件刚好出现在martym的收件箱顶端

- 15:03 jennifer注意到了事故中的Shakespeare-discuss@邮件列表

- 15:04　martym　找到了新的韵文全文，同时开始寻找更新索引的文档

- 15:06 docbrown发现崩溃现象在所有集群的所有任务中都一样，根据应用程序日志进行进一步调查

- 15:07 martym 找到了文档，开始进行Corpus更新的准备工作

- 15:10 martym 将新的韵文加入，开始进行索引任务

- 15:12　docbrown　联系了clarac & agoogler（莎士比亚搜索开发团队），帮助检查代码中的潜在问题

- 15:18 clarac 在日志中发现了文件描述符耗尽这个问题，确认了如果搜索词不存在则会导致文件描述符泄露

- 15:20 martym 的索引MapReduce任务完成

- 15:21 jennifer和docbrown 决定大幅提升实例数量，以便降低每个实例的负载，从而使它们可以在崩溃重启之前处理更多工作

- 15:23 docbrown 将所有流量导向了 USA-2 集群，允许其他集群增加副本数量

- 15:25 martym 开始将新的索引复制到其他集群

- 15:28 docbrown开始将集群实例数量扩展为2倍

- 15:32 jennifer 将负载均衡调整回正常，将流量导向了其他集群

- 15:33 其他集群的任务开始崩溃，出现了同样症状

- 15:34 发现了白板计算中关于实例副本实例增加的一个数量级错误

- 15:36 jennifer将负载均衡重新调整为USA-2，准备再增加5倍全球容量（总计10倍）

- **15:36** 故障缓解，更新过的索引复制到了全部集群

- 15:39 docbrown 启动了第二批副本数量更新，提升到了10倍容量

- 15:41 jennifer 将1%流量重新分布在所有集群中

- 15:43 其他集群的HTTP 500保持稳定，任务崩溃率很低

- 15:45 jennifer 将10% 流量重新分布在所有集群中

- 15:47 其他集群的HTTP 500稳定在SLO范围内，没有任务故障发生

- 15:50 30% 流量重新分配
- 15:55 50% 流量重新分配
- 16:00 故障结束，所有流量重新分布于所有集群中
- 16:30 应急事故管理结束，30分钟内恢复正常运行，已经达到了退出条件

其他支持信息[6]

监控页面，http://monitor/shakespeare?end_time=20151021T160000&duration=7200

[1]事故影响是指对用户以及收入等的影响。

[2]根源问题指对问题发生环境的解释。这里通常使用例如 5 Whys（参见文献[Ohn88]）这类技术来理解各种相关因素。

[3]应激反射类的AI通常太极端，实现成本太高，决策需要在更高的范围内进行。而且，还存在对某个特定问题过度优化的问题，在更可靠的方案（类似单元测试）可以预防的情况下还增加了过于具体的监控与警报方案。

[4]这一项是用来解释“差点出事”的事故的，如“山羊传送器正好也适用于其他的动物，尽管没有经过检验。”

[5]这里收集了事故的全过程，用应急事故管理文档中的时间线来填充事后总结的时间线，另外再加入一些其他相关的记录。

[6]有用的信息、文档链接、日志、屏幕截图、图表、IRC记录、IM记录等。

附录E 发布协调检查列表

这是Google最初的发布协调检查列表，写于2005年，有简单修改。

架构

- 架构草图，服务器类型，客户端请求类型
- 编程性客户端的请求

物理机与数据中心

- 物理机数量与带宽数量，数据中心，N＋2冗度，网络QoS
- 新的域名，DNS负载均衡

流量预估、容量以及性能

- HTTP流量与带宽预估，发布时的峰值，流量的组成，6个月的预测
- 压力测试，端到端测试，每个数据中心最高延迟下的容量
- 对其他我们关注的服务的影响
- 存储容量

系统可靠性与灾难恢复

- 当下列情况发生时，服务会怎么样：
  - 物理机故障，机柜故障，集群故障
  - 两个数据中心之间的网络故障

- 对每种需要联系其他服务器（后端）的服务器来说：

– 如何检测后端故障，后端故障如何处理

– 如何在不影响客户端和用户的情况下重启服务器

– 负载均衡，速度限制，超时，重试，以及错误处理

- 数据备份/恢复，灾难恢复

监控与服务器管理

- 监控内部状态，监控端到端行为，警报的管理

- 监控监控系统

- 有关财务的警报和日志

- 在集群环境下运行服务的技巧

- 不要在代码中给自己发送海量邮件，会导致邮件服务器崩溃

安全

- 安全设计评审，安全代码评审，垃圾邮件风险，验证，SSL

- 发布之前的可见/可访问性控制，各种类型的黑名单

自动化与人工任务

- 更新服务器、数据，配置文件的方式和变更管理

- 发布流程，可重复的构建过程，金丝雀测试，分阶段发布

增长问题

- 空余容量，10倍增长，增长型的警报

- 扩展性的瓶颈，线性扩展，与硬件性能的同步扩展，所需要的变更

- 缓存，数据分片/重新分片

外部依赖

- 第三方系统，监控，网络条件，流量配比，发布时的流量峰值
- 优雅降级，如何避免意外对第三方服务造成过载
- 与合作伙伴、邮件系统，以及Google内部服务良好对接

发布时间与发布计划

- 不可改变的截止日期，外部事件，星期一或者星期五
- 该服务标准的运维流程，以及其他服务的运维流程

附录F 生产环境会议记录示范

日期：2015-10-23

参与者：agoogler、clarac、docbrown、jennifer和martym

公告

- 大型事故（#465），造成错误预算耗尽

之前的待办事项评审

- 确保山羊传送器可以用于传送奶牛（bug 1011101）

— 质子加速中的非线性特质可以预知了，应该可以在几天内解决准确性问题

事故回顾

- 新韵文的发现（事故465）

— 12.1亿个请求在连锁故障与潜伏性Bug（搜索结果为空时，文件描述符泄露）的共同作用下丢失，索引中不存在新的韵文和未预料的流量

— 文件描述符的Bug已经修复（bug 5554825），已经部署到生产环境

— 调研使用flux capacitor 进行负载均衡（bug 5554823），利用负载抛弃来预防再发生（bug 5554826）

— 错误预算已经耗尽，生产环境的更新将会停止一个月。除非docbrown能够以该极为罕见、不可预知为理由获得管理层批准（但是大

家一致认为这不太可能）

紧急警报回顾

● AnnotataionConsistencyTooEventual：本周报警5次，可能是由于Bigtable跨区域的复制延迟导致

– 调查仍在进行，见Bug 4821600

– 最近不会有修复，会提高阈值以减少无效警报的产生

非紧急警报回顾

• 没有

监控系统修改/静音

● AnnotationConsistencyTooEventual，可以接受的延迟阈值从60s提升到180s，见Bug 4821600,TODO（martym）

计划中的生产变更

● USA-1 集群预计在2015-10-29 到 2015-11-02期间下线维护– 没有需要采取的动作，流量会自动切换到其他集群

资源

• 处理新韵文事故时借用了一些资源，会在下周下线多余的容量以退还容量

• 目前的利用率是 CPU 60%、RAM 75%、DISK 44%（比上周的40%、70%、40%要高）

关键服务指标

● OK 99百分比延迟：88ms □ 100ms SLO目标（过去30天）

● BAD 可用性：86.95% □ 99.99% SLO目标（过去30天）

讨论/项目更新

- 项目Molière 下两周发布

新的待办事项

- TODO（martym）：提高AnnotationConsistencyTooEventual的阈值
- TODO（docbrown）：将实例数量复原，退还资源

参考文献

[Ada15] Bram Adams, Stephany Bellomo, Christian Bird, Tamara Marshall-Keim, Foutse Khomh, and Kim Moir, "The Practice and Future of Release Engineering: A Roundtable with Three Release Engineers" (*http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=434819*), *IEEE Software*, vol. 32, no. 2 (March/April 2015), pp. 42–49.

[Agu10] M. K. Aguilera, "Stumbling over Consensus Research: Misunderstandings and Issues" (*http://dl.acm.org/citation.cfm?id=2172342*), in *Replication*, Lecture Notes in Computer Science 5959, 2010.

[All10] J. Allspaw and J. Robbins, *Web Operations: Keeping the Data on Time*: O'Reilly, 2010.

[All12] J. Allspaw, "Blameless PostMortems and a Just Culture" (*https://codeascraft.com/2012/05/22/blameless-postmortems/*), blog post, 2012.

[All15] J. Allspaw, "Trade-Offs Under Pressure: Heuristics and Observations of Teams Resolving Internet Service Outages" (*http://lup.lub.lu.se/student-papers/record/8084520/file/8084521.pdf*), MSc thesis, Lund University, 2015.

[Ana07] S. Anantharaju, "Automating web application security testing" (*https://googleonlinesecurity.blogspot.com/2007/07/automating-web-application-security.html*), blog post, July 2007.

[Ana13] R. Ananatharayan et al., "Photon: Fault-tolerant and Scalable Joining of Continuous Data Streams" (*https://research.google.com/pubs/pub41318.html*), in *SIGMOD '13*, 2013.

[And05] A. Andrieux, K. Czajkowski, A. Dan, et al., "Web Services Agreement Specification (WS-Agreement)" (*http://www.ogf.org/documents/GFD.107.pdf*), September 2005.

[Bai13] P. Bailis and A. Ghodsi, "Eventual Consistency Today: Limitations, Extensions, and Beyond" (*http://dl.acm.org/citation.cfm?id=2462076*), in *ACM Queue*, vol. 11, no. 3, 2013.

[Bai83] L. Bainbridge, "Ironies of Automation" (*http://dx.doi.org/10.1016/0005-1098(83)90046-8*), in *Automatica*, vol. 19, no. 6, November 1983.

[Bak11] J. Baker et al., "Megastore: Providing Scalable, Highly Available Storage for Interactive Services" (*https://research.google.com/pubs/pub36971.html*), in *Proceedings of the Conference on Innovative Data System Research*, 2011.

[Bar11] L. A. Barroso, "Warehouse-Scale Computing: Entering the Teenage Decade" (*http://dl.acm.org/citation.cfm?id=2019527*), talk at 38th Annual Symposium on Computer Architecture, video available online, 2011.

[Bar13] L. A. Barroso, J. Clidaras, and U. Hölzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition* (*https://research.google.com/pubs/pub41606.html*), Morgan & Claypool, 2013.

[Ben12] C. Bennett and A. Tseitlin, "Chaos Monkey Released Into The Wild" (*http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html*), blog post, July 2012.

[Bla14] M. Bland, "Goto Fail, Heartbleed, and Unit Testing Culture" (*http://martinfowler.com/articles/testing-culture.html*), blog post, June 2014.

[Boc15] L. Bock, *Work Rules!* (*https://www.workrules.net*), Twelve Books, 2015.

[Bol11] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li, "Paxos Replicated State Machines as the Basis of a High-Performance Data Store" (*https://www.usenix.org/legacy/event/nsdi11/tech/full_papers/Bolosky.pdf*), in *Proc. NSDI 2011*, 2011.

[Boy13] P. G. Boysen, "Just Culture: A Foundation for Balanced Accountability and Patient Safety" (*http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3776518/*), in *The Ochsner Journal*, Fall 2013.

[Bra15] VM Brasseur, "Failure: Why it happens & How to benefit from it" (*https://youtu.be/DLn4fZsZsKM?t=29m05s*), YAPC 2015.

[Bre01] E. Brewer, "Lessons From Giant-Scale Services" (*http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=939450*), in *IEEE Internet Computing*, vol. 5, no. 4, July / August 2001.

[Bre12] E. Brewer, "CAP Twelve Years Later: How the "Rules" Have Changed" (*http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6133253*), in *Computer*, vol. 45, no. 2, February 2012.

[Bro15] M. Brooker, "Exponential Backoff and Jitter" (*http://www.awsarchitectureblog.com/2015/03/backoff.html*), on *AWS Architecture Blog*, March 2015.

[Bro95] F. P. Brooks Jr., "No Silver Bullet—Essence and Accidents of Software Engineering", in *The Mythical Man-Month*, Boston: Addison-Wesley, 1995, pp. 180–186.

[Bru09] J. Brutlag, "Speed Matters" (*http://googleresearch.blogspot.com/2009/06/speed-matters.html*), on *Google Research Blog*, June 2009.

[Bul80] G. M. Bull, *The Dartmouth Time-sharing System*: Ellis Horwood, 1980.

[Bur99] M. Burgess, *Principles of Network and System Administration*: Wiley, 1999.

[Bur06] M. Burrows, "The Chubby Lock Service for Loosely-Coupled Distributed Systems" (*https://research.google.com/archive/chubby.html*), in *OSDI '06: Seventh Symposium on Operating System Design and Implementation*, November 2006.

[Bur16] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes" (*http://dl.acm.org/citation.cfm?id=2898444*) in *ACM Queue*, vol. 14, no. 1, 2016.

[Cas99] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance" (*http://www.pmg.lcs.mit.edu/papers/osdi99.pdf*), in *Proc. OSDI 1999*, 1999.

[Cha10] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. Henry, R. Bradshaw, and N. Weizenbaum, "FlumeJava: Easy, Efficient Data-Parallel Pipelines" (*http://research.google.com/pubs/pub35650.html*), in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010.

[Cha96] T. D. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems" (*http://dl.acm.org/citation.cfm?id=226647*), in *J. ACM*, 1996.

[Cha07] T. Chandra, R. Griesemer, and J. Redstone, "Paxos Made Live—An Engineering Perspective" (*http://research.google.com/archive/paxos_made_live.html*), in *PODC '07: 26th ACM Symposium on Principles of Distributed Computing*, 2007.

[Cha06] F. Chang et al., "Bigtable: A Distributed Storage System for Structured Data" (*https://research.google.com/archive/bigtable.html*), in *OSDI '06: Seventh Symposium on Operating System Design and Implementation*, November 2006.

[Chr09] G. P. Chrousous, "Stress and Disorders of the Stress System" (*http://www.ncbi.nlm.nih.gov/pubmed/19488073*), in *Nature Reviews Endocrinology*, vol 5., no. 7, 2009.

[Clos53] C. Clos, "A Study of Non-Blocking Switching Networks" (*http://dx.doi.org/10.1002/j.1538-7305.1953.tb01433.x*), in *Bell System Technical Journal*, vol. 32, no. 2, 1953.

[Con15] C. Contavalli, W. van der Gaast, D. Lawrence, and W. Kumari, "Client Subnet in DNS Queries" (*https://tools.ietf.org/html/draft-vandergaast-edns-client-subnet*), *IETF Internet-Draft*, 2015.

[Con63] M. E. Conway, "Design of a Separable Transition-Diagram Compiler" (*http://dl.acm.org/citation.cfm?id=366704*), in *Commun. ACM* 6, 7 (July 1963), 396–408.

[Con96] P. Conway, "Preservation in the Digital World" (*http://www.clir.org/pubs/reports/conway2/index.html*), report published by the Council on Library and Information Resources, 1996.

[Coo00] R. I. Cook, "How Complex Systems Fail" (*http://web.mit.edu/2.75/resources/random/How%20Complex%20Systems%20Fail.pdf*), in *Web Operations*: O'Reilly, 2010.

[Cor12] J. C. Corbett et al., "Spanner: Google's Globally-Distributed Database" (*https://research.google.com/archive/spanner.html*), in *OSDI '12: Tenth Symposium on Operating System Design and Implementation*, October 2012.

[Cra10] J. Cranmer, "Visualizing code coverage" (*https://quetzalcoatal.blogspot.com/2010/03/visualizing-code-coverage.html*), blog post, March 2010.

[Dea13] J. Dean and L. A. Barroso, "The Tail at Scale" (*http://research.google.com/pubs/pub40801.html*), in *Communications of the ACM*, vol. 56, 2013.

[Dea04] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters" (*https://research.google.com/archive/mapreduce.html*), in *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, December 2004.

[Dea07] J. Dean, "Software Engineering Advice from Building Large-Scale Distributed Systems" (*https://static.googleusercontent.com/media/research.google.com/en//people/jeff/stanford-295-talk.pdf*), Stanford CS297 class lecture, Spring 2007.

[Dek02] S. Dekker, "Reconstructing human contributions to accidents: the new view on error and performance" (*http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.411.4985&rep=rep1&type=pdf*), in *Journal of Safety Research*, vol. 33, no. 3, 2002.

[Dek14] S. Dekker, *The Field Guide to Understanding "Human Error"*, 3rd edition: Ashgate, 2014.

[Dic14] C. Dickson, "How Embracing Continuous Release Reduced Change Complexity" (*http://usenix.org/conference/ures14west/summit-program/presentation/dickson*), presentation at USENIX Release Engineering Summit West 2014, video available online.

[Dur05] J. Durmer and D. Dinges, "Neurocognitive Consequences of Sleep Deprivation" (*http://www.ncbi.nlm.nih.gov/pubmed/15798944*), in *Seminars in Neurology*, vol. 25, no. 1, 2005.

[Eis16] D. E. Eisenbud et al., "Maglev: A Fast and Reliable Software Network Load Balancer" (*https://research.google.com/pubs/pub44824.html*), in NSDI '16: 13th USENIX Symposium on Networked Systems Design and Implementation, March 2016.

[Ere03] J. R. Erenkrantz, "Release Management Within Open Source Projects" (*http://www.erenkrantz.com/Geeks/Research/Publications/ReleaseManagement.pdf*), in *Proceedings of the 3rd Workshop on Open Source Software Engineering*, Portland, Oregon, May 2003.

[Fis85] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process" (*http://dl.acm.org/citation.cfm?id=214121*), *J. ACM*, 1985.

[Fit12] B. W. Fitzpatrick and B. Collins-Sussman, *Team Geek: A Software Developer's Guide to Working Well with Others*: O'Reilly, 2012.

[Flo94] S. Floyd and V. Jacobson, "The Synchronization of Periodic Routing Messages" (*http://dl.acm.org/citation.cfm?id=187045*), in IEEE/ACM Transactions on Networking, vol. 2, issue 2, April 1994, pp. 122–136.

[For10] D. Ford et al, "Availability in Globally Distributed Storage Systems" (*http://research.google.com/pubs/pub36737.html*), in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[Fox99] A. Fox and E. A. Brewer, "Harvest, Yield, and Scalable Tolerant Systems" (*http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=798396*), in *Proceedings of the 7th Workshop on Hot Topics in Operating Systems*, Rio Rico, Arizona, March 1999.

[Fow08] M. Fowler, "GUI Architectures" (*http://martinfowler.com/eaaDev/uiArchs.html*), blog post, 2006.

[Gal78] J. Gall, *SYSTEMANTICS: How Systems Really Work and How They Fail*, 1st ed., Pocket, 1977.

[Gal03] J. Gall, *The Systems Bible: The Beginner's Guide to Systems Large and Small*, 3rd ed., General Systemantics Press/Liberty, 2003.

[Gaw09] A. Gawande, *The Checklist Manifesto: How to Get Things Right*: Henry Holt and Company, 2009.

[Ghe03] S. Ghemawat, H. Gobioff, and S-T. Leung, "The Google File System" (*https://research.google.com/archive/gfs.html*), in *19th ACM Symposium on Operating Systems Principles*, October 2003.

[Gil02] S. Gilbert and N. Lynch, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services" (*http://dl.acm.org/citation.cfm?id=564601*), in *ACM SIGACT News*, vol. 33, no. 2, 2002.

[Gla02] R. Glass, *Facts and Fallacies of Software Engineering*, Addison-Wesley Professional, 2002.

[Gol14] W. Golab et al., "Eventually Consistent: Not What You Were Expecting?" (*http://dl.acm.org/citation.cfm?id=2582994*), in *ACM Queue*, vol. 12, no. 1, 2014.

[Gra09] P. Graham, "Maker's Schedule, Manager's Schedule" (*http://paulgraham.com/makersschedule.html*), blog post, July 2009.

[Gup15] A. Gupta and J. Shute, "High-Availability at Massive Scale: Building Google's Data Infrastructure for Ads" (*https://research.google.com/pubs/pub44686.html*), in *Workshop on Business Intelligence for the Real Time Enterprise*, 2015.

[Ham07] J. Hamilton, "On Designing and Deploying Internet-Scale Services" (*https://www.usenix.org/legacy/event/lisa07/tech/hamilton.html*), in *Proceedings of the 21st Large Installation System Administration Conference*, November 2007.

[Han94] S. Hanks, T. Li, D. Farinacci, and P. Traina, "Generic Routing Encapsulation over IPv4 networks" (*https://tools.ietf.org/html/rfc1702*), *IETF Informational RFC*, 1994.

[Hic11] M. Hickins, "Tape Rescues Google in Lost Email Scare" (*http://blogs.wsj.com/digits/2011/03/01/tape-rescues-google-in-lost-email-scare/*), in *Digits*, *Wall Street Journal*, 1 March 2011.

[Hix15a] D. Hixson, "Capacity Planning" (*https://www.usenix.org/publications/login/feb15/capacity-planning*), in *;login:*, vol. 40, no. 1, February 2015.

[Hix15b] D. Hixson, "The Systems Engineering Side of Site Reliability Engineering" (*https://www.usenix.org/publications/login/june15/hixson*), in *;login:* vol. 40, no. 3, June 2015.

[Hod13] J. Hodges, "Notes on Distributed Systems for Young Bloods" (*https://www.somethingsimilar.com/2013/01/14/notes-on-distributed-systems-for-young-bloods/*), blog post, 14 January 2013.

[Hol14] L. Holmwood, "Applying Cardiac Alarm Management Techniques to Your On-Call" (*http://fractio.nl/2014/08/26/cardiac-alarms-and-ops/*), blog post, 26 August 2014.

[Hum06] J. Humble, C. Read, D. North, "The Deployment Production Line", in *Proceedings of the IEEE Agile Conference*, July 2006.

[Hum10] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*: Addison-Wesley, 2010.

[Hun10] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for Internet-scale systems" (*https://www.usenix.org/legacy/events/atc10/tech/full_papers/Hunt.pdf*), in *USENIX ATC*, 2010.

[IAEA12] International Atomic Energy Agency, "Safety of Nuclear Power Plants: Design, SSR-2/1" (*http://www-pub.iaea.org/MTCD/publications/PDF/Pub1534_web.pdf*), 2012.

[Jai13] S. Jain et al., "B4: Experience with a Globally-Deployed Software Defined WAN" (*https://research.google.com/pubs/pub41761.html*), in *SIGCOMM '13*.

[Jon15] C. Jones, T. Underwood, and S. Nukala, "Hiring Site Reliability Engineers" (*https://www.usenix.org/publications/login/june15/hiring-site-reliability-engineers*), in *;login:*, vol. 40, no. 3, June 2015.

[Jun07] F. Junqueira, Y. Mao, and K. Marzullo, "Classic Paxos vs. Fast Paxos: Caveat Emptor" (*http://dl.acm.org/citation.cfm?id=1323158*), in *Proc. HotDep '07*, 2007.

[Jun11] F. P. Junqueira, B. C. Reid, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems." (*http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5958223&tag=1*), in *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference* on 27 Jun 2011: 245–256.

[Kah11] D. Kahneman, *Thinking, Fast and Slow*: Farrar, Straus and Giroux, 2011.

[Kar97] D. Karger et al., "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web" (*http://dl.acm.org/citation.cfm?id=258660*), in *Proc. STOC '97*, 29th annual ACM symposium on theory of computing, 1997.

[Kem11] C. Kemper, "Build in the Cloud: How the Build System Works" (*https://google-engtools.blogspot.com/2011/08/build-in-cloud-how-build-system-works.html*), *Google Engineering Tools* blog post, August 2011.

[Ken12] S. Kendrick, "What Takes Us Down?" (*http://usenix.org/publications/login/october-2012-volume-37-number-5/what-takes-us-down*), in *;login:*, vol. 37, no. 5, October 2012.

[Kinc09] Kincaid, Jason. "T-Mobile Sidekick Disaster: Danger's Servers Crashed, And They Don't Have A Backup." *Techcrunch*. n.p., 10 Oct. 2009. Web. 20 Jan. 2015, *http://techcrunch.com/2009/10/10/t-mobile-sidekick-disaster-microsofts-servers-crashed-and-they-dont-have-a-backup*.

[Kin15] K. Kingsbury, "The trouble with timestamps" (*http://www.aphyr.com/posts/299-the-trouble-with-timestamps*), blog post, 2013.

[Kir08] J. Kirsch and Y. Amir, "Paxos for System Builders: An Overview" (*http://dl.acm.org/citation.cfm?id=1529979*), in *Proc. LADIS '08*, 2008.

[Kla12] R. Klau, "How Google Sets Goals: OKRs" (*https://library.gv.com/how-google-sets-goals-okrs-a1f69b0b72c7*), blog post, October 2012.

[Kle06] D. V. Klein, "A Forensic Analysis of a Distributed Two-Stage Web-Based Spam Attack" (*https://www.usenix.org/legacy/event/lisa06/tech/klein/klein_html/index.html*), in *Proceedings of the 20th Large Installation System Administration Conference*, December 2006.

[Kle14] D. V. Klein, D. M. Betser, and M. G. Monroe, "Making *Push On Green* a Reality" (*https://www.usenix.org/publications/login/october-2014-vol-39-no-5/making-push-green-reality*), in *;login:*, vol. 39, no. 5, October 2014.

[Kra08] T. Krattenmaker, "Make Every Meeting Matter" (*https://hbr.org/2008/02/make-every-meeting-matter*), in *Harvard Business Review*, February 27, 2008.

[Kre12] J. Kreps, "Getting Real About Distributed System Reliability" (*http://blog.empathybox.com/post/19574936361/getting-real-about-distributed-system-reliability*), blog post, 19 March 2012.

[Kri12] K. Krishan, "Weathering The Unexpected" (*http://dl.acm.org/citation.cfm?id=2366332*), in *Communications of the ACM*, vol. 55, no. 11, November 2012.

[Kum15] A. Kumar et al., "BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing" (*https://research.google.com/pubs/pub43838.html*), in *SIGCOMM '15*.

[Lam98] L. Lamport, "The Part-Time Parliament" (*http://research.microsoft.com/en-us/um/people/lamport/pubs/lamport-paxos.pdf*), in *ACM Transactions on Computer Systems 16, 2*, May 1998.

[Lam01] L. Lamport, "Paxos Made Simple" (*http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf*), in *ACM SIGACT News* 121, December 2001.

[Lam06] L. Lamport, "Fast Paxos" (*http://research.microsoft.com/pubs/64624/tr-2005-112.pdf*), in *Distributed Computing* 19.2, October 2006.

[Lim14] T. A. Limoncelli, S. R. Chalup, and C. J. Hogan, *The Practice of Cloud System Administration: Designing and Operating Large Distributed Systems, Volume 2*: Addison-Wesley, 2014.

[Loo10] J. Loomis, "How to Make Failure Beautiful: The Art and Science of Postmortems", in *Web Operations*: O'Reilly, 2010.

[Lu15] H. Lu et al, "Existential Consistency: Measuring and Understanding Consistency at Facebook" (*http://sigops.org/sosp/sosp15/current/2015-Monterey/printable/240-lu.pdf*), in *SOSP* '15, 2015.

[Mao08] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: Building Efficient Replicated State Machines for WANs" (*https://www.usenix.org/legacy/events/osdi08/tech/full_papers/mao/mao.pdf*), in *OSDI '08*, 2008.

[Mas43] A. H. Maslow, "A Theory of Human Motivation", in *Psychological Review* 50(4), 1943.

[Mau15] B. Maurer, "Fail at Scale" (*http://dl.acm.org/citation.cfm?id=2839461*), in *ACM Queue*, vol. 13, no. 12, 2015.

[May09] M. Mayer, "*This site may harm your computer* on every search result?!?!" (*https://googleblog.blogspot.com/2009/01/this-site-may-harm-your-computer-on.html*), blog post, January 2009.

[McI86] M. D. McIlroy, "A Research Unix Reader: Annotated Excerpts from the Programmer's Manual, 1971–1986" (*http://www.cs.dartmouth.edu/~doug/reader.pdf*).

[McN13] D. McNutt, "Maintaining Consistency in a Massively Parallel Environment" (*https://www.usenix.org/conference/ucms13/summit-program/presentation/mcnutt*), presentation at USENIX Configuration Management Summit 2013, video available online.

[McN14a] D. McNutt, "Accelerating the Path from Dev to DevOps" (*https://www.usenix.org/system/files/login/articles/05_mcnutt.pdf*), in *;login:*, vol. 39, no. 2, April 2014.

[McN14b] D. McNutt, "The 10 Commandments of Release Engineering" (*https://www.youtube.com/watch?v=RNMjYV_UsQ8*), presentation at 2nd International Workshop on Release Engineering 2014, April 2014.

[McN14c] D. McNutt, "Distributing Software in a Massively Parallel Environment" (*https://www.usenix.org/conference/lisa14/conference-program/presentation/mcnutt*), presentation at USENIX LISA 2014, video available online.

[Mic03] Microsoft TechNet, "What is SNMP?", last modified March 28, 2003, *https://technet.microsoft.com/en-us/library/cc776379%28v=ws.10%29.aspx*.

[Mea08] D. Meadows, *Thinking in Systems*: Chelsea Green, 2008.

[Men07] P. Menage, "Adding Generic Process Containers to the Linux Kernel" (*https://www.kernel.org/doc/ols/2007/ols2007v2-pages-45-58.pdf*), in *Proc. of Ottawa Linux Symposium*, 2007.

[Mer11] N. Merchant, "Culture Trumps Strategy, Every Time" (*https://hbr.org/2011/03/culture-trumps-strategy-every*), in *Harvard Business Review*, March 22, 2011.

[Moc87] P. Mockapetris, "Domain Names - Implementation and Specification" (*https://tools.ietf.org/html/rfc1035*), *IETF Internet Standard*, 1987.

[Mol86] C. Moler, "Matrix Computation on Distributed Memory Multiprocessors", in *Hypercube Multiprocessors 1986*, 1987.

[Mor12a] I. Moraru, D. G. Andersen, and M. Kaminsky, "Egalitarian Paxos" (*http://www.pdl.cmu.edu/PDL-FTP/associated/CMU-PDL-12-108.pdf*), *Carnegie Mellon University Parallel Data Lab Technical Report CMU-PDL-12-108*, 2012.

[Mor14] I. Moraru, D. G. Andersen, and M. Kaminsky, "Paxos Quorum Leases: Fast Reads Without Sacrificing Writes" (*http://dl.acm.org/citation.cfm?id=2671001*), in *Proc. SOCC '14*, 2014.

[Mor12b] J. D. Morgenthaler, M. Gridnev, R. Sauciuc, and S. Bhansali, "Searching for Build Debt: Experiences Managing Technical Debt at Google" (*https://research.google.com/pubs/pub37755.html*), in *Proceedings of the 3rd Int'l Workshop on Managing Technical Debt*, 2012.

[Nar12] C. Narla and D. Salas, "Hermetic Servers" (*http://googletesting.blogspot.com/2012/10/hermetic-servers.html*), blog post, 2012.

[Nel14] B. Nelson, "The Data on Diversity" (*http://dl.acm.org/citation.cfm?id=2684442.2597886*), in *Communications of the ACM*, vol. 57, 2014.

[Nic12] K. Nichols and V. Jacobson, "Controlling Queue Delay" (*http://dl.acm.org/citation.cfm?id=2209336*), in *ACM Queue*, vol. 10, no. 5, 2012.

[Oco12] P. O'Connor and A. Kleyner, *Practical Reliability Engineering*, 5th edition: Wiley, 2012.

[Ohn88] T. Ohno, *Toyota Production System: Beyond Large-Scale Production*: Productivity Press, 1988.

[Ong14] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm (Extended Version)" (*https://ramcloud.stanford.edu/raft.pdf*).

[Pen10] D. Peng and F. Dabek, "Large-scale Incremental Processing Using Distributed Transactions and Notifications" (*https://research.google.com/pubs/pub36726.html*), in *Proc. of the 9th USENIX Symposium on Operating System Design and Implementation*, November 2010.

[Per99] C. Perrow, *Normal Accidents: Living with High-Risk Technologies*, Princeton University Press, 1999.

[Per07] A. R. Perry, "Engineering Reliability into Web Sites: Google SRE" (*https://research.google.com/pubs/pub32583.html*), in *Proc. of LinuxWorld 2007*, 2007.

[Pik05] R. Pike, S. Dorward, R. Griesemer, S. Quinlan, "Interpreting the Data: Parallel Analysis with Sawzall" (*https://research.google.com/archive/sawzall.html*), in *Scientific Programming Journal* vol. 13, no. 4, 2005.

[Pot16] R. Potvin and J. Levenberg, "The Motivation for a Monolithic Codebase: Why Google stores billions of lines of code in a single repository", in *Communications of the ACM*, forthcoming July 2016. Video available on YouTube (*https://www.youtube.com/watch?v=W71BTkUbdqE*).

[Roo04] J. J. Rooney and L. N. Vanden Heuvel, "Root Cause Analysis for Beginners" (*http://asq.org/quality-progress/2004/07/quality-tools/root-cause-analysis-for-beginners.html*), in *Quality Progress*, July 2004.

[Sai39] A. de Saint Exupéry, *Terre des Hommes* (Paris: Le Livre de Poche, 1939, in translation by Lewis Galantière as *Wind, Sand and Stars*.

[Sam14] R. R. Sambasivan, R. Fonseca, I. Shafer, and G. R. Ganger, "So, You Want To Trace Your Distributed System? Key Design Insights from Years of Practical Experience" (*http://pdl.cmu.edu/PDL-FTP/SelfStar/CMU-PDL-14-102_abs.shtml*), Carnegie Mellon University Parallel Data Lab Technical Report CMU-PDL-14-102, 2014.

[San11] N. Santos and A. Schiper, "Tuning Paxos for High-Throughput with Batching and Pipelining" (*http://rd.springer.com/chapter/10.1007%2F978-3-642-25959-3_11*), in *13th Int'l Conf. on Distributed Computing and Networking*, 2012.

[Sar97] N. B. Sarter, D. D. Woods, and C. E. Billings, "Automation Surprises", in *Handbook of Human Factors & Ergonomics*, 2nd edition, G. Salvendy (ed.), Wiley, 1997.

[Sch14] E. Schmidt, J. Rosenberg, and A. Eagle, *How Google Works* (*http://www.howgoogleworks.net*): Grand Central Publishing, 2014.

[Sch15] B. Schwartz, "The Factors That Impact Availability, Visualized" (*https://www.vividcortex.com/blog/the-factors-that-impact-availability-visualized*), blog post, 21 December 2015.

[Sch90] F. B. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial" (*http://dl.acm.org/citation.cfm?id=98167*), in *ACM Computing Surveys*, vol. 22, no. 4, 1990.

[Sec13] Securities and Exchange Commission, "Order In the Matter of Knight Capital Americas LLC" (*https://www.sec.gov/litigation/admin/2013/34-70694.pdf*), file 3-15570, 2013.

[Sha00] G. Shao, F. Berman, and R. Wolski, "Master/Slave Computing on the Grid" (*http://www.cs.ucsb.edu/~rich/publications/shao-hcw.pdf*), in *Heterogeneous Computing Workshop*, 2000.

[Shu13] J. Shute et al., "F1: A Distributed SQL Database That Scales" (*https://research.google.com/pubs/pub41344.html*), in *Proc. VLDB 2013*, 2013.

[Sig10] B. H. Sigelman et al., "Dapper, a Large-Scale Distributed Systems Tracing Infrastructure" (*https://research.google.com/pubs/pub36356.html*), Google Technical Report, 2010.

[Sin15] A. Singh et al., "Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network" (*https://research.google.com/pubs/pub43837.html*), in *SIGCOMM '15*.

[Skel13] M. Skelton, "Operability can Improve if Developers Write a Draft Run Book" (*http://blog.softwareoperability.com/2013/10/16/operability-can-improve-if-developers-write-a-draft-run-book/*), blog post, 16 October 2013.

[Slo11] B. Treynor Sloss, "Gmail back soon for everyone" (*http://gmailblog.blogspot.com/2011/02/gmail-back-soon-for-everyone.html*), blog post, 28 February 2011.

[Tat99] S. Tatham, "How to Report Bugs Effectively" (*http://www.chiark.greenend.org.uk/~sgtatham/bugs.html*), 1999.

[Ver15] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg" (*https://research.google.com/pubs/pub43438.html*), in *Proceedings of the European Conference on Computer Systems*, 2015.

[Wal89] D. R. Wallace and R. U. Fujii, "Software Verification and Validation: An Overview" (*http://www-usr.inf.ufsm.br/~ceretta/papers/fujii89_software_vv.pdf*), *IEEE Software*, vol. 6, no. 3 (May 1989), pp. 10, 17.

[War14] R. Ward and B. Beyer, "BeyondCorp: A New Approach to Enterprise Security" (*https://www.usenix.org/publications/login/dec14/ward*), in *;login:*, vol. 39, no. 6, December 2014.

[Whi12] J. A. Whittaker, J. Arbon, and J. Carollo, *How Google Tests Software*: Addison-Wesley, 2012.

[Woo96] A. Wood, "Predicting Software Reliability" (*http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=544240*), in *Computer*, vol. 29, no. 11, 1996.

[Wri12a] H. K. Wright, "Release Engineering Processes, Their Faults and Failures" (*http://www.hyrumwright.org/papers/dissertation.pdf*), (section 7.2.2.2) PhD Thesis, University of Texas at Austin, 2012.

[Wri12b] H. K. Wright and D. E. Perry, "Release Engineering Practices and Pitfalls" (*http://www.hyrumwright.org/papers/icse2012.pdf*), in *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. (IEEE, 2012), pp. 1281–1284.

[Wri13] H. K. Wright, D. Jasper, M. Klimek, C. Carruth, Z. Wan, "Large-Scale Automated Refactoring Using ClangMR" (*http://static.googleusercontent.com/media/research.google.com/en/us/pubs/archive/41342.pdf*), in *Proceedings of the 29th International Conference on Software Maintenance (ICSM '13)*, (IEEE, 2013), pp. 548–551.

[Zoo14] ZooKeeper Project (Apache Foundation), "ZooKeeper Recipes and Solutions" (*http://zookeeper.apache.org/doc/trunk/recipes.html*), in ZooKeeper 3.4 documentation, 2014.

索引 [1]

Symbols

- /varz HTTP handler, 109

A

- abusive client behavior, 382
- access control, 89
- ACID datastore semantics, 287, 341
- acknowledgments, xxi-xxiv
- adaptive throttling, 250
- Ads Database, 73-75
- AdSense, 31
- aggregate availability equation, 27, 477
- aggregation, 114, 180
- agility vs. stability, 97
  - (see also software simplicity)
- Alertmanager service, 119
- alerts
  - defined, 56
  - false-positive, 180
  - software for, 18
    - (see also Borgmon; time-series monitoring)
- anacron, 316
- Apache Mesos, 15
- App Engine, 146
- archives vs. backups, 343
- asynchronous distributed consensus, 289
- atomic broadcast systems, 295
- attribution policy, xx
- automation
  - applying to cluster turnups, 75-81
  - vs. autonomous systems, 67
  - benefits of, 67-70
  - best practices for change management, 10
  - Borg example, 81
  - cross-industry lessons, 467
  - database example, 73-75
  - Diskerase example, 85
  - focus on reliability, 83
  - Google's approach to, 70
  - hierarchy of automation classes, 72
  - recommendations for enacting, 84
  - specialized application of, 79
  - use cases for, 70-73
- automation tools, 194
- autonomous systems, 67
- Auxon case study, 207-209, 211-213
- availability, 38, 341
  - (see also service availability)
- availability table, 477

B

- B4 network, 15
- backend servers, 19, 231
- backends, fake, 204
- backups (see data integrity)
- Bandwidth Enforcer (BwE), 17
- barrier tools, 193, 195, 293
- batch processing pipelines, 350
- batching, 282, 302, 329
- Bazel, 90
- best practices
  - capacity planning, 482
  - for change management, 10
  - error budgets, 481
  - failures, 479
  - feedback, 174
  - for incident management, 166

monitoring, 481
overloads and failure, 483
postmortems, 171-172, 482
reward systems, 174
role of release engineers in, 88
rollouts, 480
service level objectives, 480
team building, 483
bibliography, 501
Big Data, 327
Bigtable, 17, 31, 65
bimodal latency, 273
black-box monitoring, 55, 59, 120
blameless cultures, 170
Blaze build tool, 90
Blobstore, 17, 342
Borg, 14-16, 81-83, 329
Borg Naming Service (BNS), 16
Borgmon, 108-123
(see also time-series monitoring)
alerting, 18, 118
configuration, 121
rate() function, 116
rules, 114-118
sharding, 119
timeseries arena, 111
vectors, 112-113
break-glass mechanisms, 201
build environments, 190
business continuity, 337
Byzantine failures, 290, 304

C

campuses, 14
canarying, 34, 155, 189, 380
CAP theorem, 286
CAPA (corrective and preventative action), 465
capacity planning
approaches to, 105
best practices for, 482
Diskerase example, 85
distributed consensus systems and, 307
drawbacks of "queries per second", 248
drawbacks of traditional plans, 208
further reading on, 106
intent-based (see intent-based capacity planning)
mandatory steps for, 11
preventing server overload with, 266
product launches and, 376
traditional approach to, 207
cascading failures
addressing, 280-283
causes of, 260-265
defined, 259, 308
factors triggering, 276
overview of, 283
preventing server overload, 265-276
testing for, 278-280
(see also overload handling)
change management, 10
(see also automation)
change-induced emergencies, 153-155
changelists (CLs), 20
Chaos Monkey, 196
checkpoint state, 195
cherry picking tactic, 89
Chubby lock service, 18, 291
planned outage, 39, 47
client tasks, 231
client-side throttling, 249
clients, 19
clock drift, 287
Clos network fabric, 14
cloud environment
data integrity strategies, 341, 356
definition of data integrity in, 340
evolution of applications in, 341
technical challenges of, 344
clusters
applying automation to turnups, 75-81
cluster management solution, 329
defined, 14
code samples, xx
cognitive flow state, 409
cold caching, 274
colocation facilities (colos), 85
Colossus, 16
command posts, 164
communication and collaboration
blameless postmortems, 171
case studies, 432-439
importance of, 440
with Outalator, 181
outside SRE team, 437
position of SRE in Google, 425
production meetings (see production meetings)

within SRE team, 430
company-wide resilience testing, 106
compensation structure, 128
computational optimization, 209
configuration management, 93, 153, 201, 277
configuration tests, 188
consensus algorithms
Egalitarian Paxos, 302
Fast Paxos, 301, 320
improving performance of, 296
Multi-Paxos, 303
Paxos, 290, 303
Raft, 298, 302
Zab, 302
(see also distributed consensus systems)
consistency
eventual, 287
through automation, 67
consistent hashing, 227
constraints, 208
Consul, 291
consumer services, identifying risk tolerance of, 28-31
continuous build and deployment
Blaze build tool, 90
branching, 90
build targets, 90
configuration management, 93
deployment, 93
packaging, 91
Rapid release system, 90, 91
testing, 90
typical release process, 92
contributors, xxii-xxiv
coroutines, 327
corporate network security, 106
correctness guarantees, 335
correlation vs. causation, 136
costs
availability targets and, 30, 32
direct, 4
of failing to embrace risk, 25
indirect, 4
of sysadmin management approach, 4
CPU consumption, 248, 262, 383
crash-fail vs. crash-recover algorithms, 290
cron
at large scale, 325
building at Google, 319-325
idempotency, 316
large-scale deployment of, 317
leader and followers, 321
overview of, 326
Paxos algorithm and, 320-325
purpose of, 315
reliability applications of, 316
resolving partial failures, 322
storing state, 324
tracking cron job state, 319
uses for, 315
cross-industry lessons
Apollo 8, xvii
comparative questions presented, 459
decision-making skills, 469-470
Google's application of, 470
industry leaders contributing, 460
key themes addressed, 459
postmortem culture, 465-467
preparedness and disaster testing, 462-465
repetitive work/operational overhead, 467
current state, exposing, 138

D

D storage layer, 16
dashboards
benefits of, 57
defined, 55
data analysis, with Outalator, 181
data integrity
backups vs. archives, 343
case studies in, 360-366
conditions leading to failure, 346
defined, 339
expanded definition of, 340
failure modes, 349
from users' perspective, 345
overview of, 368
selecting strategy for, 341-343, 356
SRE approach to, 349-360
SRE objectives for, 344-349
SRE principles applied to, 367-368
strict requirements, 340
technical challenges of, 344
data processing pipelines
business continuity and, 337
challenges of uneven work distribution, 328
challenges to periodic pattern, 328
drawbacks of periodic, 329-332

effect of big data on, 328
monitoring problems, 331-332
origin of, 327
overview of, 338
pipeline depth, 328
simple vs. multiphase pipelines, 328
Workflow system, 333, 335
data recovery, 359
datacenters
backbone network for, 15
data validation, 357
load balancing, 231-246
topology of, 14
datastores
ACID and BASE, 287, 341, 347
reliable replicated, 292
Decider, 74
decision-making skills, 469
defense in depth, for data integrity, 349, 361, 367
demand forecasting, 11
dependency hierarchies, 58, 263
deployment, 93
(see also continuous build and deployment)
development environment, 19
development/ops split, 4
DevOps, 7
Direct Server Response (DSR), 228
disaster recovery tools, 195
disaster role playing, 401
disaster testing, 462-465
Disaster and Recovery Testing (DiRT), 462
disk access, 303
Diskerase process, 85
distractibility, 411
distributed consensus systems
benefits of, 285
coordination, use in, 293
deploying, 304-312
locking, use in, 286
monitoring, 312
need for, 285
overview of, 313
patterns for, 291-295
performance of, 296-304
principles, 289
quorum composition, 310
quorum leasing technique, 299
(see also consensus algorithms)
distributed periodic scheduling (see cron)
DNS (Domain Name System)
EDNS0 extension, 225
load balancing using, 224-227
DoubleClick for Publishers (DFP), 437-439
drains, 277
DTSS communication files, 327
dueling proposers situation, 298
durability, 38

E

early detection for data integrity, 356
(see also data integrity)
Early Engagement Model, 448-451
"embarrassingly parallel" algorithms, 328
embedded engineers, 417-423
emergency preparedness, 361
cross-industry lessons, 462
emergency response
change-induced emergencies, 153-155
essential elements of, 151
Five Whys, 140, 487
guidelines for, 10
initial response, 151
lessons learned, 158
overview of, 159
process-induced emergencies, 155
solution availability, 158
test-induced emergencies, 152
encapsulation, 228
endpoints, in debugging, 138
engagements (see SRE engagement model)
error budgets
benefits of, 35
best practices for, 481
forming, 34
guidelines for, 8
motivation for, 33
error rates, 38, 60
Escalator, 178
ETL pipelines, 327
eventual consistency, 287
executor load average, 253

F

failures, best practices for, 479
(see also cascading failures)
fake backends, 204
false-positive alerts, 180

feature flag frameworks, 381
file descriptors, 263
Five Whys, 140, 487
flow control, 233
FLP impossibility result, 290
Flume, 328
fragmentation, 229

G

gated operations, 89
Generic Routing Encapsulation (GRE), 228
GFE (Google Frontend), 21, 232
GFS (Google File System), 76, 293, 318-319, 354
global overload, 248
Global Software Load Balancer (GSLB), 18
Gmail, 65, 360
Google Apps for Work, 29
Google Compute Engine, 38
Google production environment
  best practices for, 479-484
  complexity of, 205
  datacenter topology, 14
  development environment, 19
  hardware, 13
  Shakespeare search service, 20-22
  software infrastructure, 19
  system software, 15-19
Google Workflow system
  as model-view-controller pattern, 334
  business continuity and, 337
  correctness guarantees, 335
  development of, 333
  stages of execution in, 335
graceful degradation, 267
GTape, 360

H

Hadoop Distributed File System (HDFS), 16
handoffs, 164
"hanging chunk" problem, 329
hardware
  managing failures, 15
  software that "organizes", 15-19
  terminology used for, 13
health checks, 281
healthcare.gov, 103
hermetic builds, 89
hierarchical quorums, 311
high-velocity approach, 24, 88
hotspotting, 236

I

idempotent operations, 78, 316
incident management
  best practices for, 166
  effective, 161
  formal protocols for, 130
  incident management process, 153, 163
  incident response, 104
  managed incident example, 165
  roles, 163
  template for, 485
  unmanaged incident example, 161
  when to declare an incident, 166
infrastructure services
  identifying risk tolerance of, 31
  improved SRE through automation, 69
integration proposals, 89
integration tests, 186, 201
intent-based capacity planning
  Auxon implementation, 211-213
  basic premise of, 209
  benefits of, 209
  defined, 209
  deploying approximation, 214
  driving adoption of, 215-217
  precursors to intent, 210
  requirements and implementation, 213
  selecting intent level, 210
  team dynamics, 218
interrupts
  cognitive flow state and, 409
  dealing with, 407
  dealing with high volumes, 412
  determining approach to handling, 408
  distractibility and, 411
  managing operational load, 408
  on-call engineers and, 412
  ongoing responsibilities, 413
  polarizing time, 411
  reducing, 413
  ticket assignments, 413
IRC (Internet Relay Chat), 164

J

jobs, 16
Jupiter network fabric, 14

L

labelsets, 112
lame duck state, 234
latency
    defined, 341
    measuring, 38
    monitoring for, 60
launch coordination
    checklist, 373-380, 493
    engineering (LCE), 370, 384-387
        (see also product launches)
lazy deletion, 349
leader election, 286, 292
lease systems, 294
Least-Loaded Round Robin policy, 243
level of service, 37
    (see also service level objectives (SLOs))
living incident documents, 164
load balancing
    datacenter
        datacenter services and tasks, 231
        flow control, 233
        Google's application of, 231
        handling overload, 247
        ideal CPU usage, 232, 248
        lame duck state, 234
        limiting connections pools, 235-240
        packet encapsulation, 228
        policies for, 240-246
        SRE software engineering dynamics, 218
    distributed consensus systems and, 307
    frontend
        optimal solutions for, 223
        using DNS, 224-227
        virtual IP addresses (VIPs), 227
    policy
        Least-Loaded Round Robin, 243
        Round Robin, 241
        Weighted Round Robin, 245
load shedding, 267
load tests, 383
lock services, 18, 293
logging, 138
Lustre, 16

M

machines
    defined, 13, 56
    managing with software, 15
majority quorums, 304
MapReduce, 328
mean time
    between failures (MTBF), 184, 199
    to failure (MTTF), 10
    to repair (MTTR), 10, 68, 184
memory exhaustion, 263
Mencius algorithm, 302
meta-software, 70
Midas Package Manager (MPM), 91
model-view-controller pattern, 334
modularity, 100
Moiré load pattern in pipelines, 331
monitoring distributed systems
    avoiding complexity in, 62
    benefits of monitoring, 56, 107
    best practices for, 481
    blackbox vs. whitebox, 59, 120
    case studies, 65-66
    challenges of, 64, 107
    change-induced emergencies, 154
    creating rules for, 63
    four golden signals of, 60
    guidelines for, 9
    instrumentation and performance, 61
    monitoring philosophy, 64
    resolution, 62
    setting expectations for, 57
    short- vs. long-term availability, 66
    software for, 18
    symptoms vs. causes, 58
    terminology, 55
    valid monitoring outputs, 10
        (see also Borgmon; time-series monitoring)
Multi-Paxos protocol, 297, 303
    (see also consensus algorithms)
multi-site teams, 127
multidimensional matrices, 112
multiphase pipelines, 328
MySQL
    migrating, 73-75, 437
    test-induced emergencies and, 152

N

N + 2 configuration, 22, 210-212, 266, 482
negative results, 144
Network Address Translation, 228
network latency, 300

network load balancer, 227
network partitions, 287
Network Quality of Service (QoS), 157, 252
network security, 106
networking, 17
NORAD Tracks Santa website, 369
number of "nines", 38, 477

O

older releases, rebuilding, 89
on-call
  balanced on-call, 127
  benefits of, 132
  best practices for, 393, 400-405
  compensation structure, 128
  continuing education, 406
  education practices, 392, 395
  formal incident-management protocols, 130
  inappropriate operational loads, 130
  initial learning experiences, 394
  learning checklists, 403
  overview of, 125, 406
  resources for, 129
  rotation schedules, 126
  shadow on-call, 405
  stress-reduction techniques, 128
  target event volume, 8
  targeted project work, 397
  team building, 391
  time requirements, 128
  training for, 395-401
  training materials, 397
  typical activities, 126
one-phase pipelines, 328
open commenting/annotation system, 171
operational load
  cross-industry lessons, 467
  managing, 408
  ongoing responsibilities, 408
  types of, 407
operational overload, 130
operational underload, 132
operational work (see toil)
out-of-band checks and balances, 342, 357
out-of-band communications systems, 154
outage tracking
  baselines and progress tracking, 177
  benefits of, 182
  Escalator, 178
  Outalator, 178-182
Outalator
  aggregation in, 180
  benefits of, 178
  building your own, 179
  incident analysis, 181
  notification process, 178
  reporting and communication, 181
  tagging in, 180
overhead, 49
overload handling
  approaches to, 247
  best practices for, 483
  client-side throttling, 249
  load from connections, 257
  overload errors, 253
  overview of, 258
  per-client retry budget, 254
  per-customer limits, 248
  per-request retry budget, 254
  product launches and, 383
  request criticality, 251
  retrying requests, 254
    (see also retries, RPC)
  utilization signals, 253
    (see also cascading failures)

P

package managers, 91
packet encapsulation, 228
Paxos consensus algorithm
  Classic Paxos algorithm, 301
  disk access and, 303
  Egalitarian Paxos consensus algorithm, 302
  Fast Paxos consensus algorithm, 301, 320
  Lamport's Paxos protocol, 290
    (see also consensus algorithms)
performance
  efficiency and, 12
  monitoring, 61
performance tests, 186
periodic pipelines, 328
periodic scheduling (see cron)
persistent storage, 303
Photon, 305
pipelining, 302
planned changes, 277
policies and procedures, enforcing, 89
post hoc analysis, 58

postmortems
benefits of, 169
best practices for, 171-174, 482
collaboration and sharing in, 171
concept of, 169
cross-industry lessons, 465-467
example postmortem, 487-490
formal review and publication of, 171
Google's philosophy for, 169
guidelines for, 8
introducing postmortem cultures, 172
on-call engineering and, 400
ongoing improvements to, 175
rewarding participation in, 174
triggers for, 170
privacy, 341
proactive testing, 159
problem reports, 136
process death, 276
process health checks, 281
process updates, 277
process-induced emergencies, 155
Prodtest (Production Test), 76
product launches
best practices for, 480
defined, 370
development of Launch Coordination Engineering (LCE), 384-387
driving convergence and simplification, 374
launch coordination checklists, 373-380, 493
launch coordination engineering, 370
NORAD Tracks Santa example, 369
overview of, 387
processes for, 372
rate of, 370
techniques for reliable, 380-384
production environment (see Google production environment)
production inconsistencies
detecting with Prodtest, 76
resolving idempotently, 78
production meetings, 426-430
agenda example, 497
production probes, 202
Production Readiness Review process (see SRE engagement model)
production tests, 187
protocol buffers (protobufs), 19, 202
Protocol Data Units, 229
provisioning, guidelines for, 11
PRR (Production Readiness Review) model, 442, 444-448
push frequency, 34
push managers, 413
Python's safe_load, 201

Q

"queries per second" model, 248
Query of Death, 276
queuing
controlled delay, 267
first-in, first-out, 267
last-in, first-out, 267
management of, 266, 294
queuing-as-work-distribution pattern, 295
quorum (see distributed consensus systems)

R

Raft consensus protocol, 298, 302
(see also consensus algorithms)
RAID, 354
Rapid automated release system, 90, 91
read workload, scaling, 298
real backups, 343
real-time collaboration, 171
recoverability, 347
recovery, 359
recovery systems, 345
recursion (see recursion)
recursive DNS servers, 225
recursive separation of responsibilities, 163
redundancy, 347, 354
Reed-Solomon erasure codes, 354
regression tests, 186
release engineering
challenges of, 87
continuous build and deployment, 90-94
defined, 87
instituting, 95
philosophy of, 88-90
the role of release engineers, 87
wider application of, 95
reliability testing
amount required, 184
benefits of, 204
break-glass mechanisms, 201
canary tests, 189

configuration tests, 188
coordination of, 197
creating test and build environments, 190
error budgets, 8, 33-35, 481
expecting test failure, 199-200
fake backend versions, 204
goals of, 183
importance of, xvi
integration tests, 186, 201
MTTR and, 184
performance tests, 186
proactive, 159
production probes, 202
production tests, 187
regression tests, 186
reliability goals, 25
sanity testing, 186
segregated environments and, 198
smoke tests, 186
speed of, 196
statistical tests, 196
stress tests, 188
system tests, 186
testing at scale, 192-204
timing of, 187
unit tests, 185
reliable replicated datastores, 292
Remote Procedure Call (RPC), 19, 138, 252
bimodal, 273
deadlines
missing, 271
propagating, 267, 272
queue management, 266, 294
selecting, 271
retries, 268-271
RPC criticality, 251
(see also overload handling)
replicas
adding, 307
drawbacks of leader replicas, 308
location of, 306, 310
number deployed, 304
replicated logs, 305
replicated state machine (RSM), 291
replication, 347, 354
request latency, 38, 60
request profile changes, 277
request success rate, 27
resilience testing, 106
resources
allocation of, 14, 16
exhaustion, 261
limits, 278
(see also capacity planning)
restores, 355
retention, 348
retries, RPC
avoiding, 254
cascading failures due to, 268
considerations for automatic, 270
diagnosing outages due to, 271
handling overload errors and, 254
per-client retry budgets, 254
per-request retry budgets, 254
reverse engineering, 398
reverse proxies, 157
revision history, 351
risk management
balancing risk and innovation, 25
costs of, 25
error budgets, 33-36, 481
key insights, 36
measuring service risk, 26
risk tolerance of services, 28-33
rollback procedures, 153
rollouts, 277, 379, 480
root cause
analysis of, 105, 169
(see also postmortems)
defined, 56
Round Robin policy, 241
round-trip-time (RTT), 300
rows, 14
rule evaluation, in monitoring systems, 114-118

S

Safari® Books Online, xxi
sanity testing, 186
saturation, 60
scale
defined, 341
issues in, 347
security
in release engineering, 89
new approach to, 106
self-service model, 88
separation of responsibilities, 163
servers

vs. clients, 19
defined, 13
overload scenario, 260
preventing overload, 265-276
service availability
availability table, 477
cost factors, 30, 32
defined, 38
target for consumer services, 29
target for infrastructure service, 31
time-based equation, 26
types of consumer service failures, 29
types of infrastructure services failures, 32
service health checks, 281
service latency
looser approach to, 31
monitoring for, 60
service level agreements (SLAs), 39
service level indicators (SLIs)
aggregating raw measurements, 41
collecting indicators, 41
defined, 38
standardizing indicators, 43
service level objectives (SLOs)
agreements in practice, 47
best practices for, 480
choosing, 37-39
control measures, 46
defined, 38
defining objectives, 43
selecting relevant indicators, 40
statistical fallacies and, 43
target selection, 45
user expectations and, 39, 46
service management
comprehensive approach to, xvi
Google's approach to, 5-7
sysadmin approach to, 3, 67
service reliability hierarchy
additional resources, 106
capacity planning, 105
development, 106
diagram of, 103
incident response, 104
monitoring, 104
product launch, 106
root cause analysis, 105
testing, 105
service unavailability, 264
Service-Oriented Architecture (SOA), 81
Shakespeare search service, example
alert, 137
applying SRE to, 20-22
cascading failure example, 259-283
debugging, 139
engagement, 283, 445
incident management, 485
postmortem, 487-490
production meeting, 497-499
sharded deployments, 307
SHEDDABLE_PLUS criticality value, 251
simplicity, 97-101
Sisyphus automation framework, 93
Site Reliability Engineering (SRE)
activities included in, 103
approach to learning, xix
basic components of, xv
benefits of, 6
challenges of, 6
defined, xiii-xiv, 5
early engineers, xvii
Google's approach to management, 5-7, 425
growth of at Google, 473, 474
hiring, 5, 391
origins of, xvi
sysadmin approach to management, 3, 67
team composition and skills, 5, 126, 473
tenets of, 7-12
typical activities of, 52
widespread applications of, xvi
slow startup, 274
smoke tests, 186
SNMP (Simple Networking Monitoring Protocol), 111
soft deletion, 350
software bloat, 99
software engineering in SRE
activities included in, 106
Auxon case study, 207-209
benefits of, 222
encouraging, 215
fostering, 218
Google's focus on, 205
importance of, 205
intent-based capacity planning, 209-218
staffing and development time, 219
team dynamics, 218
software fault tolerance, 34

software simplicity
avoiding bloat, 99
modularity, 100
predictability and, 98
release simplicity, 100
reliability and, 101
source code purges, 98
system stability versus agility, 97
writing minimal APIs, 99
Spanner, 17, 32, 337
SRE engagement model
aspects addressed by, 443
Early Engagement Model, 448-451
frameworks and platforms in, 451-455
importance of, 441
Production Readiness Review, 442, 444-448
SRE tools
automation tools, 194
barrier tools, 193, 195
disaster recovery tools, 195
testing, 193
writing, 202
SRE Way, 12
stability vs. agility, 97
(see also software simplicity)
stable leaders, 302
statistical tests, 196
storage stack, 16
stress tests, 188
strong leader process, 297
Stubby, 19
subsetting
defined, 235
deterministic, 238
process of, 235
random, 237
selecting subsets, 236
synchronous consensus, 289
sysadmins (systems administrators), 3, 67
system software
managing failures with, 15
managing machines, 15
storage, 16
system tests, 186
system throughput, 38
systems administrators (sysadmins), 3, 67
systems engineering, 390

T

tagging, 180
"task overloaded" errors, 253
tasks
backend, 231
client, 231
defined, 16
TCP/IP communication protocol, 300
team building
benefits of Google's approach to, 6, 473
best practices for, 483
development focus, 6
dynamics of SRE software engineering, 218
eliminating complexity, 98
engineering focus, 6, 7, 52, 126-127, 474
multi-site teams, 127
self-sufficiency, 88
skills needed, 5
staffing and development time, 219
team composition, 5
terminology (Google-specific)
campuses, 14
clients, 19
clusters, 14
datacenters, 14
frontend/backend, 19
jobs, 16
machines, 13
protocol buffers (protobufs), 19
racks, 14
rows, 14
servers, 13, 19
tasks, 16
test environments, 190
(see also reliability testing)
test-induced emergencies, 152
testing (see reliability testing)
text logs, 138
thread starvation, 263
throttling
adaptive, 250
client-side, 249
"thundering herd" problems, 331, 383
time-based availability equation, 26, 477
Time-Series Database (TSDB), 112
time-series monitoring
alerting, 118
black-box monitoring, 120
Borgmon monitoring system, 108

collection of exported data, 110
instrumentation of applications, 109
maintaining Borgmon configuration, 121
monitoring topology, 119
practical approach to, 108
rule evaluation, 114-118
scaling, 122
time-series data storage, 111-113
tools for, 108
time-to-live (TTL), 225
timestamps, 292
toil
benefits of limiting, 24, 51
calculating, 51
characteristics of, 49
cross-industry lessons, 467
defined, 49
drawbacks of, 52
vs. engineering work, 52
traffic analysis, 21-22, 60
training practices, 392, 395-397
triage process, 137
Trivial File Transfer Protocol (TFTP), 157
troubleshooting
App Engine case study, 146-149
approaches to, 133
common pitfalls, 135
curing issues, 145
diagnosing issues, 139-142
examining system components, 138
logging, 138
model of, 134
pitfalls, 135-136
problem reports, 136
process diagram, 135
simplifying, 150
systematic approach to, 150
testing and treating issues, 142-145
triage, 137
turndown automation, 157, 277
typographical conventions, xix

U

unit tests, 185
UNIX pipe, 327
unplanned downtime, 26
uptime, 341
user requests
criticality values assigned to, 251
job and data organization, 22
monitoring failures, 60
request latency, 38
request latency monitoring, 60
retrying, 254
servicing of, 21
success rate metrics, 27
traffic analysis, 22, 60
utilization signals, 253

V

variable expressions, 113
vectors, 112
velocity, 341
Viceroy project, 432-437
virtual IP addresses (VIPs), 227

W

"War Rooms", 164
Weighted Round Robin policy, 245
Wheel of Misfortune exercise, 173
white-box monitoring, 55, 59, 108
workloads, 296

Y

yield, 38
YouTube, 29

Z

Zab consensus, 302
Zookeeper, 291

[1]：索引所列页码为本书英文版页码，请参照正文侧边用“ ”表示的原书页码。

关于编著者

Betsy Beyer是Google 纽约负责SRE的一名技术文档作家。她之前曾为遍布全球的Google数据中心与Mountain View硬件运维团队编写文档。在搬到纽约之前，Betsy是Stanford大学技术性写作课程的讲师。她曾经学习国际关系与英文文学，并在Stanford和Tulane获得学历。

Chris Jones是Google App Engine的一名SRE。Google App Engine是一个PaaS服务，每天处理超过280亿个请求。他的办公室在旧金山，他之前的工作包括Google广告统计、数据仓库，以及用户支持系统的维护。在之前，Chris曾经在学校IT行业任职，同时参与过竞选数据分析，以及一些BSD内核的修改。他有计算机工程、经济学，以及技术政策学的学位。同时他也是一名有执照的职业工程师。

Jennifer Petoff是Google SRE团队的一名项目经理，工作地点在都柏林，爱尔兰。她曾经负责管理大型全球项目，包括：科学研究、工程、人力资源，以及广告等。Jennifer在加入Google之前，曾在化工行业任职八年。她获得了Stanford大学的化学博士与学士学位，同时她还拥有Rochester大学的心理学学位。

Niall Murphy是Google爱尔兰团队广告SRE的负责人。他拥有20年互联网行业经验，目前是INEX（爱尔兰网络互联枢纽）的主席。他曾经写作以及参与写作很多科技文章与书籍，包括O'Reilly 出版的IPv6 Network Administration，以及很多RFC。他目前在参与书写爱尔兰互联网发展史。他拥有计算机科学、数学，以及诗歌学的学历（他当时一定是想错了！）。他目前与妻子和两个儿子居住在都柏林。

封面介绍

封面上的动物是Ornate巨蜥，一种西非和中非居住的两栖动物。在1997年之前，它都被认为是Nile巨蜥的一个子类，但是由于它身上不同的花纹，现在被认为是Varanus stellatus 与 Varanus niloticus 的一个变种。

同时，它的活动范围也比Nile巨蜥要小，倾向于居住在雨林低地上。

Ornate 巨蜥是巨型蜥蜴，它可以生长到1.8～2.1米长。它们与Nile巨蜥相比，颜色更加明亮，从肩膀到尾巴都有着橄榄色的皮肤以及一些明亮的黄色斑点。和其他巨蜥类似，这种动物也有肌肉粗壮的身躯、锋利的爪子，和细长的头。它们的鼻孔很高，允许它们在水中长时间停留。它们是出色的游泳选手和登山健将，这使得它们能够以鱼、蛙、蛋、昆虫，以及小型哺乳动物为食。

巨蜥经常被作为宠物饲养，但它们需要精心照料，因此并不适合初养者。当它们感到受到威胁时可能是很危险的（用强有力的尾巴抽打、搔抓或咬），但有可能通过定期驯养以及教育它们并与美味食物结合起来，在一定程度上驯服它们。

许多出现在O'Reilly图书封面上的动物都是濒危动物；所有这些动物对世界都很重要。要了解更多有关如何帮助它们的知识，可访问animals.oreilly.com。